



Programmez avec le langage C++

Par Mathieu Nebra (M@teo21)
et Matthieu Schaller (Nanoc)



www.siteduzero.com

*Licence Creative Commons BY-NC-SA 2.0
Dernière mise à jour le 25/04/2012*

Sommaire

Sommaire	2
Lire aussi	9
Programmez avec le langage C++	11
Partie 1 : [Théorie] Découverte de la programmation en C++	12
Qu'est-ce que le C++ ?	12
Les programmes	12
Les langages de programmation	13
Le C++ face aux autres langages	14
Le C++ : langage de haut ou de bas niveau ?	14
Résumé des forces du C++	15
Petit aperçu du C++	16
La petite histoire du C++	16
De l'Algol au C++	17
Le concepteur	17
En résumé	17
Les logiciels nécessaires pour programmer	18
Les outils nécessaires au programmeur	19
Les projets	19
Choisissez votre IDE	20
Code::Blocks (Windows, Mac OS, Linux)	20
Télécharger Code::Blocks	20
Créer un nouveau projet	23
Visual C++ (Windows seulement)	25
Installation	26
Créer un nouveau projet	27
Ajouter un nouveau fichier source	29
La fenêtre principale de Visual	30
Xcode (Mac OS seulement)	31
Xcode, où es-tu ?	31
Lancement	32
Nouveau projet	32
Compilation	34
Votre premier programme	37
Le monde merveilleux de la console	37
Les programmes graphiques	37
Les programmes console	38
Notre première cible : les programmes console	39
Création et lancement d'un premier projet	39
Création d'un projet	39
Lancement du programme	40
Explications du premier code source	41
include	42
using namespace	42
int main()	43
cout	43
return	44
Commentez vos programmes !	44
Les différents types de commentaires	45
Commentons notre code source !	45
Utiliser la mémoire	47
Qu'est-ce qu'une variable ?	47
Les noms de variables	47
Les types de variables	48
Déclarer une variable	49
Le cas des strings	52
Une astuce pour gagner de la place	52
Déclarer sans initialiser	52
Afficher la valeur d'une variable	54
Les références	56
Une vraie calculatrice	58
Demander des informations à l'utilisateur	59
Lecture depuis la console	59
Une astuce pour les chevrons	60
D'autres variables	60
Le problème des espaces	61
Demander d'abord la valeur de pi	62
Modifier des variables	63
Changer le contenu d'une variable	63
Une vraie calculatrice de base !	65
Les constantes	66
Déclarer une constante	67
Un premier exercice	67
Les raccourcis	69
L'incréméntation	69
La décréméntation	70

Les autres opérations	70
Encore plus de maths !	71
L'en-tête cmath	71
Quelques autres fonctions présentes dans cmath	72
Le cas de la fonction puissance	73
Les structures de contrôle	75
Les conditions	75
La condition if	75
La condition switch	79
Booléens et combinaisons de conditions	80
Les booléens	81
Combiner des conditions	81
Les boucles	83
La boucle while	83
La boucle do ... while	84
La boucle for	85
Découper son programme en fonctions	87
Créer et utiliser une fonction	87
Présentation des fonctions	87
Définir une fonction	88
Une fonction toute simple	89
Appeler une fonction	90
Plusieurs paramètres	91
Pas d'arguments	91
Des fonctions qui ne renvoient rien	91
Quelques exemples	92
Le carré	92
Réutiliser la même fonction	93
Une fonction à deux arguments	93
Passage par valeur et passage par référence	95
Passage par valeur	95
Passage par référence	96
Avancé: Le passage par référence constante	98
Utiliser plusieurs fichiers	99
Les fichiers nécessaires	99
Déclarer la fonction dans les fichiers	103
Documenter son code	105
Des valeurs par défaut pour les arguments	106
Les valeurs par défaut	107
Cas particuliers, attention danger	109
Règles à retenir	110
Les tableaux	112
Les tableaux statiques	112
Un exemple d'utilisation	112
Déclarer un tableau statique	113
Accéder aux éléments d'un tableau	114
Parcourir un tableau	115
Un petit exemple	116
Les tableaux et les fonctions	117
Les tableaux dynamiques	118
Déclarer un tableau dynamique	119
Accéder aux éléments d'un tableau	120
Changer la taille	120
Retour sur l'exercice	122
Les vector et les fonctions	123
Les tableaux multi-dimensionnels	123
Déclaration d'un tableau multi-dimensionnel	124
Accéder aux éléments	125
Aller plus loin	125
Les strings comme tableaux	125
Accéder aux lettres	125
Les fonctions	126
Lire et écrire des fichiers	127
Écrire dans un fichier	128
L'en-tête fstream	128
Ouvrir un fichier en écriture	128
Écrire dans un flux	129
Les différents modes d'ouverture	130
Lire un fichier	131
Ouvrir un fichier en lecture	131
... et le lire	131
Lire un fichier en entier	132
Quelques astuces	133
Fermer prématurément un fichier	133
Le curseur dans le fichier	134
Connaître sa position	135
Se déplacer	135
Connaître la taille d'un fichier	136
[TP] Le mot mystère	137
Préparatifs et conseils	138
Principe du jeu "Le mot mystère"	138
Quelques conseils pour bien démarrer	139

Un peu d'aide pour mélanger les lettres	140
Correction	142
Le code	142
Des explications	143
Téléchargement	144
Aller plus loin	144
Les pointeurs	145
Une question d'adresse	146
Afficher l'adresse	147
Les pointeurs	148
Déclarer un pointeur	148
Stocker une adresse	150
Afficher l'adresse	151
Accéder à la valeur pointée	152
Récapitulatif de la notation	152
L'allocation dynamique	153
La gestion automatique de la mémoire	153
Allouer un espace mémoire	153
Libérer la mémoire	155
Un exemple complet	156
Quand utiliser des pointeurs	157
Partager une variable	157
Choisir parmi plusieurs éléments	158
Partie 2 : [Théorie] La Programmation Orientée Objet	160
Introduction : la vérité sur les strings enfin dévoilée	161
Des objets... pour quoi faire ?	161
Ils sont beaux, ils sont frais mes objets	161
Imaginez... un objet	161
L'horrible secret du type string	164
Pour un ordinateur, les lettres n'existent pas	164
Les textes sont des tableaux de char	165
Créer et utiliser des objets string	166
Créer un objet string	166
Concaténation de chaînes	168
Comparaison de chaînes	169
Opérations sur les string	170
Attributs et méthodes	170
Quelques méthodes utiles du type string	170
Les classes (Partie 1/2)	173
Créer une classe	174
Créer une classe, oui mais laquelle ?	175
Bon, on la crée cette classe ?	175
Ajout de méthodes et d'attributs	176
Droits d'accès et encapsulation	178
Les droits d'accès	181
L'encapsulation	182
Séparer prototypes et définitions	183
Personnage.h	184
Personnage.cpp	185
main.cpp	188
Les classes (Partie 2/2)	189
Constructeur et destructeur	190
Le constructeur	190
Le destructeur	194
Les méthodes constantes	195
Associer des classes entre elles	196
La classe Arme	197
Adapter la classe Personnage pour utiliser une Arme	198
Action !	201
Prototype et include	202
Implémentation	202
Appel de afficherEtat dans le main	202
Méga schéma résumé	204
La surcharge d'opérateurs	206
Petits préparatifs	206
Qu'est-ce que c'est ?	206
La classe Duree pour nos exemples	206
Les opérateurs arithmétiques (+, -, *, /, %)	208
Mode d'utilisation	208
Les opérateurs raccourcis	209
Implémentation de +=	210
Quelques tests	211
Télécharger le projet	213
Bonus track #1	213
Bonus track #2	214
Les autres opérateurs arithmétiques	215
Les opérateurs de flux (<<)	216
Définir ses propres opérateurs pour cout	216
Implémentation d'operator<<	217
Ouf ! Maintenant dans le main, que du bonheur !	219
Les opérateurs de comparaison (==, >, <, ...)	219
L'opérateur ==	219

L'opérateur !=	221
L'opérateur <	221
Les autres opérateurs de comparaison	222
TP: La POO en pratique avec ZFraction	223
Préparatifs et conseils	224
Description de la classe ZFraction	224
Créer un nouveau projet	225
Le code de base des fichiers	226
Choix des attributs de la classe	227
Les constructeurs	227
Les opérateurs	228
Simplifier les fractions	228
Correction	229
Les constructeurs	229
Afficher une fraction	229
L'opérateur d'addition	230
L'opérateur de multiplication	231
Les opérateurs de comparaison	231
Les opérateurs d'ordre	232
Simplifier les fractions	233
Code complet	234
Aller plus loin	238
Classes et pointeurs	239
Pointeur d'une classe vers une autre classe	240
Gestion de l'allocation dynamique	241
Allocation de mémoire pour l'objet	242
Désallocation de mémoire pour l'objet	243
N'oubliez pas que m_arme est maintenant un pointeur !	244
Le pointeur this	244
Le constructeur de copie	246
Le problème	246
Création du constructeur de copie	248
Le constructeur de copie une fois terminé	250
L'opérateur d'affectation	250
L'héritage	252
Exemple d'héritage simple	253
Comment reconnaître un héritage ?	253
Notre exemple : la classe Personnage	253
La classe Guerrier hérite de la classe Personnage	254
La classe Magicien hérite aussi de Personnage	256
La dérivation de type	258
Héritage et constructeurs	260
Appeler le constructeur de la classe mère	261
Transmission de paramètres	261
Schéma résumé	262
La portée protected	263
Le masquage	264
Une fonction de la classe mère	264
La fonction est héritée dans les classes filles	265
Le masquage	266
Économiser du code	267
Le démasquage	267
Le polymorphisme	269
La résolution des liens	269
La résolution statique des liens	270
La résolution dynamique des liens	271
Les fonctions virtuelles	271
Déclarer une méthode virtuelle...	271
... et utiliser une référence	272
Les méthodes spéciales	273
Le cas des constructeurs	274
Le cas du destructeur	274
Le code amélioré	274
Les collections hétérogènes	276
Le retour des pointeurs	276
Utiliser la collection	276
Les fonctions virtuelles pures	279
Le problème des roues	279
Les classes abstraites	280
Éléments statiques et amitié	282
Les méthodes statiques	282
Créer une méthode statique	282
Quelques exemples de l'utilité des méthodes statiques	283
Les attributs statiques	283
Créer un attribut statique dans une classe	283
L'amitié	285
Qu'est-ce que l'amitié ?	285
Retour sur la classe Duree	285
Déclarer une fonction amie d'une classe	287
L'amitié et la responsabilité	288
Partie 3 : [Pratique] Créez vos propres fenêtres avec Qt	289
Introduction à Qt	290

Dis papa, comment on fait des fenêtres ?	290
Un mot de vocabulaire à connaître : GUI	290
Les différents moyens de créer des GUI	291
Présentation de Qt	293
Plus fort qu'une bibliothèque : un framework	293
Qt est multiplateforme	293
L'histoire de Qt	295
La licence de Qt	295
Qui utilise Qt ?	295
Installation de Qt	296
Télécharger Qt	296
Installation sous Windows	296
Qt Creator	298
Compiler votre première fenêtre Qt	299
Présentation de Qt Creator	300
Création d'un projet Qt vide	300
Ajout d'un fichier main.cpp	304
Codons notre première fenêtre !	306
Le code minimal d'un projet Qt	306
Affichage d'un widget	307
Un peu de théorie : Qt et la compilation	308
La compilation "normale", sans Qt	309
La compilation particulière avec Qt	309
Diffuser le programme	310
Personnaliser les widgets	312
Modifier les propriétés d'un widget	313
Les accesseurs avec Qt	313
Quelques exemples de propriétés des boutons	314
Qt et l'héritage	319
De l'héritage en folie	319
QObject : une classe de base incontournable	320
Les classes abstraites	322
Un widget peut en contenir un autre	322
Contenant et contenu	322
Créer une fenêtre contenant un bouton	323
Tout widget peut en contenir d'autres	325
Des includes "oubliés"	326
Hériter un widget	327
Edition des fichiers	328
La destruction automatique des widgets enfants	331
Compilation	332
Les signaux et les slots	333
Le principe des signaux et slots	334
Connexion d'un signal à un slot simple	335
Le principe de la méthode connect()	336
Utilisation de la méthode connect() pour quitter	336
Utilisation de la méthode connect() pour afficher "A propos"	337
Des paramètres dans les signaux et slots	339
Dessin de la fenêtre	339
Connexion avec des paramètres	340
Exercice	342
Créer ses propres signaux et slots	342
Créer son propre slot	342
Créer son propre signal	346
Les boîtes de dialogue usuelles	348
Afficher un message	349
Quelques rappels et préparatifs	349
Ouvrir une boîte de dialogue avec une méthode statique	350
Personnaliser les boutons de la boîte de dialogue	353
Récupérer la valeur de retour de la boîte de dialogue	355
Saisir une information	356
Saisir un texte (QInputDialog::getText)	356
Saisir un entier (QInputDialog::getInteger)	358
Saisir un nombre décimal (QInputDialog::getDouble)	359
Choix d'un élément parmi une liste (QInputDialog::getItem)	360
Sélectionner une police	361
Sélectionner une couleur	363
Sélection d'un fichier ou d'un dossier	365
Sélection d'un dossier existant (QFileDialog::getExistingDirectory)	366
Ouverture d'un fichier (QFileDialog::getOpenFileName)	366
Enregistrement d'un fichier (QFileDialog::getSaveFileName)	368
Apprendre à lire la documentation de Qt	370
Où trouver la doc ?	370
Avec internet : sur le site de Nokia	370
Sans internet : avec Qt Assistant	372
Les différentes sections de la doc	373
API Lookup	373
Qt Topics	374
Comprendre la documentation d'une classe	374
Introduction	375
Public Types	376
Properties	376

Public Functions	377
Public Slots	379
Signals	379
Protected Functions	380
Additional Inherited Members	380
Detailed description	380
Positionner ses widgets avec les layouts	381
Le positionnement absolu et ses défauts	382
Le code Qt de base	382
Les défauts du positionnement absolu	383
L'architecture des classes de layout	384
Les layouts horizontaux et verticaux	385
Le layout horizontal	385
Résumé du code	387
Résultat	387
Schéma des conteneurs	388
Le layout vertical	388
La suppression automatique des widgets	389
Le layout de grille	390
Schéma de la grille	390
Utilisation basique de la grille	390
Un widget qui occupe plusieurs cases	392
Le layout de formulaire	394
Combiner les layouts	396
Un cas concret	396
Utilisation de addLayout	397
Exercice	398
Les principaux widgets	400
Les fenêtres	400
Quelques rappels sur l'ouverture d'une fenêtre	400
Quelques classes particulières pour les fenêtres	402
Une fenêtre avec QWidget	403
Une fenêtre avec QDialog	405
Les boutons	407
QPushButton : un bouton	408
QCheckBox : une case à cocher	409
QRadioButton : les boutons radio	410
Les afficheurs	411
QLabel : afficher du texte ou une image	411
QProgressBar : une barre de progression	414
Les champs	414
QLineEdit : champ de texte à une seule ligne	415
QTextEdit : champ de texte à plusieurs lignes	415
QSpinBox : champ de texte de saisie d'entiers	416
QDoubleSpinBox : champ de texte de saisie de nombres décimaux	416
QSlider : un curseur pour sélectionner une valeur	417
QComboBox : une liste déroulante	417
Les conteneurs	418
QFrame : une bordure	418
QGroupBox : un groupe de widgets	420
QTabWidget : des pages d'onglets	421
TP : ZeroClassGenerator	425
Notre objectif	425
Un générateur de classe C++	425
Quelques conseils techniques	426
Correction	428
main.cpp	428
FenPrincipale.h	428
FenPrincipale.cpp	429
FenCodeGenere.h	432
FenCodeGenere.cpp	432
Télécharger le projet	434
Des idées d'améliorations	434
La fenêtre principale	438
Présentation de QMainWindow	438
Structure de la QMainWindow	438
Exemple de QMainWindow	439
Le code de base	440
La zone centrale (SDI et MDI)	441
Définition de la zone centrale (type SDI)	443
Définition de la zone centrale (type MDI)	444
Les menus	448
Créer un menu pour la fenêtre principale	448
Création d'actions pour les menus	449
Les sous-menus	451
Manipulations plus avancées des QAction	452
La barre d'outils	454
Ajouter une action	454
Ajouter un widget	455
Ajouter un séparateur	456
Plus d'options pour la barre d'outils	457
Les docks	458
Créer un QDockWidget	458

Peupler le QDockWidget	459
Code complet	459
La barre d'état	461
Les messages temporaires	461
Les messages normaux et permanents	462
Les status tips des QAction	463
Exercice : créer un éditeur de texte	463
Traduire son programme avec Qt Linguist	465
Les étapes de la traduction	465
Préparer son code à une traduction	466
Utilisez QString pour manipuler des chaînes de caractères	466
Faites passer les chaînes à traduire par la méthode tr()	466
Créer les fichiers de traduction .ts	468
Traduire l'application sous Qt Linguist	470
Lancer l'application traduite	472
Compiler le .ts en .qm	472
Charger un fichier de langue .qm dans l'application	473
Modéliser ses fenêtres avec Qt Designer	476
Présentation de Qt Designer	477
Choix du type de fenêtre à créer	478
Analyse de la fenêtre de Qt Designer	480
Placer des widgets sur la fenêtre	482
Placer les widgets de manière absolue	482
Utiliser les layouts	483
Insérer des spacers	484
Editer les propriétés des widgets	485
Configurer les signaux et les slots	487
Utiliser la fenêtre dans votre application	490
Notre nouvel exemple	490
Le principe de la génération du code source	491
Utiliser la fenêtre dans notre application	492
Personnaliser le code et utiliser les Auto-Connect	494
TP : zNavigo, le navigateur web des Zéros !	496
Les navigateurs et les moteurs web	497
Les principaux navigateurs	497
Le moteur web	498
Les principaux navigateurs et leurs moteurs	500
Configurer son projet pour utiliser WebKit	501
Organisation du projet	502
Objectif	502
Les fichiers du projet	503
Utiliser QWebView pour afficher une page web	503
La navigation par onglets	504
Let's go !	505
Génération de la fenêtre principale	505
main.cpp	506
FenPrincipale.h (première version)	506
Construction de la fenêtre	507
Méthodes de gestion des onglets	510
Les slots personnalisés	511
FenPrincipale.h (version complète)	511
Implémentation des slots	512
Conclusion et améliorations possibles	515
Télécharger le code source et l'exécutable	516
Améliorations possibles	516
L'architecture MVC avec les widgets complexes	517
Présentation de l'architecture MVC	518
L'architecture simplifiée modèle/vue de Qt	519
Les classes gérant le modèle	519
Les classes gérant la vue	520
Appliquer un modèle à la vue	521
Plusieurs modèles ou plusieurs vues	522
Utilisation d'un modèle simple	524
Le modèle appliqué à un QTreeView	525
Le modèle appliqué à un QListView	526
Le modèle appliqué à un QTableView	527
Utilisation de modèles personnalisables	528
QStringListModel : une liste de chaînes de caractères QString	528
QStandardItemModel : une liste à plusieurs niveaux et plusieurs colonnes	529
Gestion des sélections	533
Une sélection unique	534
Une sélection multiple	535
Communiquer en réseau avec son programme	537
Comment communique-t-on en réseau ?	538
1/ L'adresse IP : identification des machines sur le réseau	539
2/ Les ports : différents moyens d'accès à un même ordinateur	540
3/ Le protocole : transmettre des données avec le même "langage"	541
L'architecture du projet de Chat avec Qt	544
Les architectures réseau	544
Principe de fonctionnement du Chat	545
Structure des paquets	547
Réalisation du serveur	548

Création du projet	548
La fenêtre du serveur	548
main.cpp	549
FenServeur.h	549
FenServeur.cpp	550
Lancement du serveur	562
Réalisation du client	563
Dessin de la fenêtre avec Qt Designer	563
Client.pro	564
main.cpp	565
FenClient.h	565
FenClient.cpp	566
Test du Chat et améliorations	573
Tester le Chat	573
Améliorations à réaliser	575
Partie 4 : [Théorie] Utilisez la bibliothèque standard	575
Qu'est-ce que la bibliothèque standard ?	576
Un peu d'histoire	576
La petite histoire raccourcie du C++	576
Pourquoi une bibliothèque standard ?	577
Le contenu de la SL	577
L'héritage du C	577
Les flux	577
La STL	578
Le reste	578
Se documenter sur la SL	578
Des ressources sur le web	578
L'héritage du C	579
L'en-tête cmath	579
L'en-tête cctype	579
L'en-tête ctime	581
L'en-tête cstdlib	582
Les autres en-têtes	583
Les conteneurs	583
Stocker des éléments	584
Les deux catégories de conteneurs	585
Quelques méthodes communes	586
Les séquences et leurs adaptateurs	586
Les vector, encore et toujours	587
Les deque, ces drôles de tableaux	588
Les stack, une histoire de pile	589
Les queue, une histoire de file	590
Les priority_queue, la fin de l'égalité	591
Les list, à voir plus tard	592
Les tables associatives	592
Les autres tables associatives	593
Choisir le bon conteneur	593
Itérateurs et foncteurs	595
Itérateurs : des pointeurs boostés	595
Déclarer un itérateur...	595
... et itérer	596
Des méthodes uniquement pour les itérateurs	596
Les différents itérateurs	597
La pleine puissance des list et map	598
La même chose pour les map	599
Foncteur : la version objet des fonctions	601
Créer un foncteur	601
Des foncteurs évolutifs	602
Les prédicats	603
Les foncteurs pré-définis	604
Fusion des deux concepts	605
Modifier le comportement d'une map	605
Récapitulatif des conteneurs les plus courants	606
La puissance des algorithmes	607
Un premier exemple	608
Un début en douceur	608
Application aux autres conteneurs	609
Compter, chercher, trier	610
Compter des éléments	610
Le retour des prédicats	611
Chercher	612
Trier !	612
Encore plus d'algos	613
Utiliser deux séries à la fois	615
Partie 5 : [Théorie] Notions avancées	617
La gestion des erreurs avec les exceptions	617
Un problème bien ennuyeux	617
Exemple d'erreur d'implémentation	617
Quelques mauvaises solutions	618
La gestion des exceptions	619
Principe général	619

Les 3 mots-clés en détail	619
La bonne solution	621
Les exceptions standards	623
La classe exception	623
Le travail pré-mâché	626
Les exceptions de vector	627
Relancer une exception	628
Les assertions	628
Claquer une assertion	628
Désactiver les assertions	629
Créer des templates	630
Les fonctions templates	631
Ce que l'on aimerait faire	631
Une première fonction template	631
Où mettre la fonction ?	634
Tous les types sont-ils utilisables ?	634
Des fonctions plus compliquées	634
La spécialisation	635
La spécialisation	636
L'ordre des fonctions	637
Les classes templates	637
Le type des attributs	637
Création de la classe	638
Les méthodes	638
Instanciation d'une classe template	640
Ce que vous pouvez encore apprendre	642
... sur le langage C++	642
L'héritage multiple	642
Les namespaces	643
Les types énumérés	645
Les typedefs	646
... sur la bibliothèque Qt	646
Module GUI : des petites fonctionnalités cachées	647
Module réseau : utilisez des classes de haut niveau	649
Module SQL : accès aux bases de données	649
Module XML : pour ceux qui doivent gérer des données au format XML	650
Module Core : toutes les fonctionnalités de base de Qt	650
D'autres bibliothèques	651
Créer des jeux en 2D	651
Faire de la 3D	652
Plus de GUI	652
Manipuler du son	653
Boost	653



Programmez avec le langage C++

Par



Mathieu Nebra (M@teo21) et



Matthieu Schaller (Nanoc)

Mise à jour : 27/05/2011

Difficulté : Difficile  Durée d'étude : 3 mois

66 915 visites depuis 7 jours, classé 5/792

La programmation C++ vous intéresse mais ça vous paraît trop compliqué ?

Ce cours de C++ est fait pour des débutants comme vous qui n'ont jamais programmé !

Le langage C++ est un des langages les plus célèbres au monde. Très utilisé, notamment dans le secteur des jeux vidéo qui apprécie ses performances et ses possibilités, le C++ est désormais incontournable pour les développeurs.

Le C++ est le descendant du [langage C](#). Ces deux langages, bien que semblables au premier abord, sont néanmoins *différents*. Le C++ propose de nouvelles fonctionnalités, comme la programmation orientée objet (POO). Elles en font un langage très puissant qui permet de programmer avec une approche différente du langage C.

Dans ce cours, nous découvrirons aussi une bibliothèque appelée Qt. Elle ajoute un très large éventail de possibilités au C++ : elle va nous permettre de créer des fenêtres et menus, mais aussi d'utiliser les fonctionnalités réseau de votre ordinateur ! 😊



Quelques programmes C++ que nous réaliserons



Ce cours vous plaît ?

Si vous avez aimé ce cours, vous pouvez retrouver le livre "*Programmez avec le langage C++*" des mêmes auteurs, en vente [sur le Site du Zéro](#), en librairie et dans les boutiques en ligne. Vous y trouverez ce cours adapté au format papier avec une série de chapitres inédits.

[Plus d'informations](#)

Partie 1 : [Théorie] Découverte de la programmation en C++

Vous souhaitez découvrir le C++ mais vous n'avez jamais fait de programmation avant ? Commencez par là !

Vous découvrirez comment fonctionne la programmation, quels logiciels il faut installer et quelles sont les techniques de base à connaître.

Qu'est-ce que le C++ ?

L'informatique vous passionne et vous aimeriez apprendre à programmer ? Et pourquoi pas après tout ! La programmation peut sembler difficile au premier abord mais c'est un univers beaucoup plus accessible qu'il n'y paraît !

Vous vous demandez sûrement par où commencer, si le C++ est fait pour vous, s'il n'est pas préférable de démarrer avec un autre langage. Vous vous demandez si vous allez pouvoir faire tout ce que vous voulez, quelles sont les forces et les faiblesses du C++ ...

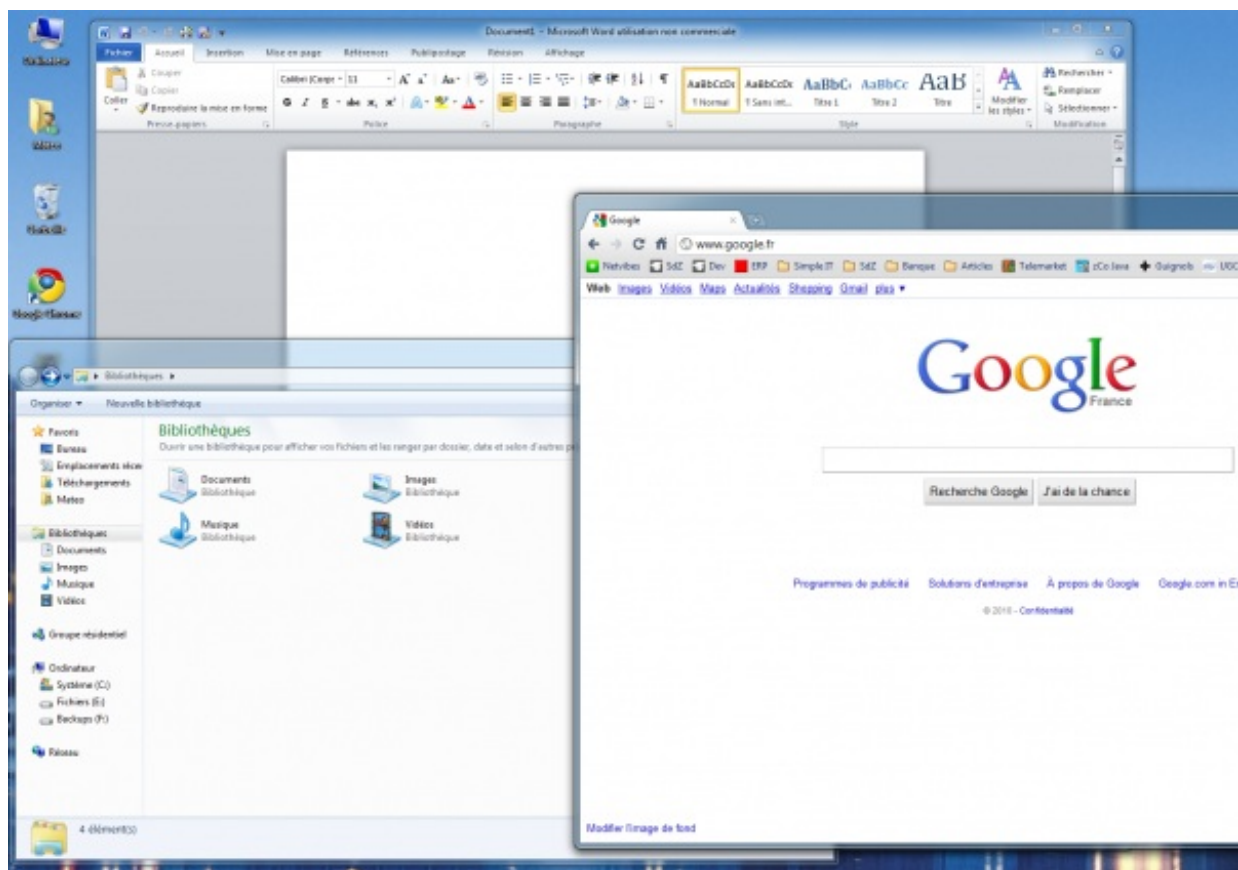
Dans ce chapitre, je vais tenter de répondre à toutes ces questions.

N'oubliez pas : c'est un *cours pour débutants*. *Aucune connaissance préalable n'est requise*. Même si vous n'avez jamais programmé de votre vie, tout ce que vous avez besoin de faire c'est de lire ce cours progressivement, sans brûler les étapes et en pratiquant régulièrement en même temps que moi !

Les programmes

Les programmes sont à la base de l'informatique. Ce sont eux qui vous permettent d'exécuter des actions sur votre ordinateur.

Prenons par exemple la figure suivante qui représente une capture d'écran de mon ordinateur. On y distingue 3 fenêtres correspondant à 3 programmes différents. Du premier plan à l'arrière-plan :



- le navigateur web Google Chrome, qui permet de consulter des sites web ;
- l'explorateur de fichiers, qui permet de gérer les fichiers sur son ordinateur ;
- le traitement de texte Microsoft Word, qui permet de rédiger lettres et documents.

Comme vous le voyez, chacun de ces programmes est conçu dans un but précis. On pourrait aussi citer les jeux, par exemple, qui sont prévus pour s'amuser : Starcraft II (figure suivante), World of Warcraft, Worms, Team Fortress 2, etc. Chacun d'eux correspond à un programme différent.



Tous les programmes ne sont pas forcément visibles. C'est le cas de ceux qui surveillent les mises à jour disponibles



pour votre ordinateur ou, dans une moindre mesure, de votre antivirus. Ils tournent tous en « tâche de fond », ils n'affichent pas toujours une fenêtre ; mais cela ne les empêche pas d'être actifs et de travailler !



Moi aussi je veux créer des programmes ! Comment dois-je m'y prendre ?

Tout d'abord, commencez par mesurer vos ambitions. Un jeu tel que Starcraft II nécessite des dizaines de développeurs à plein temps, pendant plusieurs années. Ne vous mettez donc pas en tête des objectifs trop difficiles à atteindre.

En revanche, si vous suivez ce cours, vous aurez de solides bases pour développer des programmes. Au cours d'un TP, nous réaliserons même notre propre navigateur web (simplifié) comme Mozilla Firefox et Google Chrome ! Vous saurez créer des programmes dotés de fenêtres. Avec un peu de travail supplémentaire, vous pourrez même créer des jeux 2D et 3D si vous le désirez. Bref, avec le temps et à force de persévérance, vous pourrez aller loin.

Alors oui, je n'oublie pas votre question : vous vous demandez comment réaliser des programmes. La programmation est un univers très riche. On utilise des *langages de programmation* qui permettent d'expliquer à l'ordinateur ce qu'il doit faire. Voyons plus en détail ce que sont les langages de programmation.

Les langages de programmation

Votre ordinateur est une machine étonnante et complexe. À la base, il ne comprend qu'un langage très simple constitué de 0 et de 1. Ainsi, un message tel que celui-ci :

```
1010010010100011010101001010111010100011010010
```

... peut signifier quelque chose comme « Affiche une fenêtre à l'écran ».



Ouah ! Mais c'est super compliqué ! On va être obligé d'apprendre ce langage ?

Heureusement non.

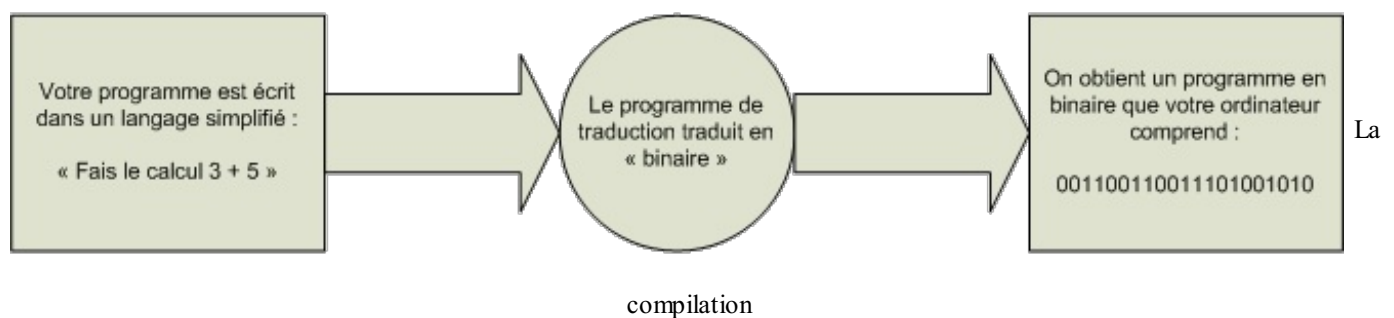
S'il fallait écrire dans ce langage (qu'on appelle *langage binaire*), il ne faudrait pas des années pour concevoir un jeu comme Starcraft II mais plutôt des millénaires (sans rire !).

Pour se simplifier la vie, les informaticiens ont créé des langages intermédiaires, plus simples que le binaire. Il existe aujourd'hui des centaines de langages de programmation. Pour vous faire une idée, vous pouvez consulter une [liste des langages de programmation sur Wikipédia](#). Chacun de ces langages a des spécificités, nous y reviendrons.

Tous les langages de programmation ont le même but : vous permettre de parler à l'ordinateur plus simplement qu'en binaire. Voici comment cela fonctionne :

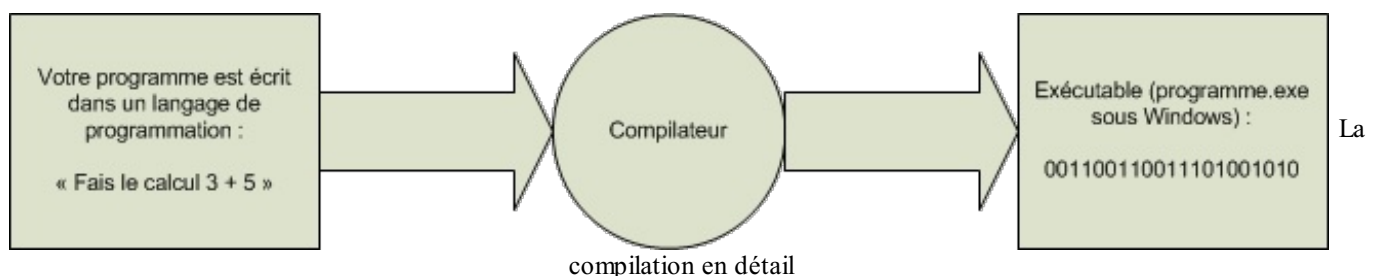
1. Vous écrivez des instructions pour l'ordinateur dans un langage de programmation (par exemple le C++) ;
2. Les instructions sont traduites en binaire grâce à un programme de « traduction » ;
3. L'ordinateur peut alors lire le binaire et faire ce que vous avez demandé !

Résumons ces étapes dans un schéma (figure suivante).



Le fameux « programme de traduction » s'appelle en réalité le *compilateur*. C'est un outil indispensable. Il vous permet de transformer votre code, écrit dans un langage de programmation, en un vrai programme exécutable.

Reprenons le schéma précédent et utilisons un vrai vocabulaire d'informaticien (figure suivante).



Voilà ce que je vous demande de retenir pour le moment : ce n'est pas bien compliqué mais c'est la base à connaître absolument !



Mais justement, comment dois-je faire pour choisir le langage de programmation que je vais utiliser ? Tu as dit toi-même qu'il en existe des centaines !
Lequel est le meilleur ? Est-ce que le C++ est un bon choix ?

Les programmeurs (aussi appelés *développeurs*) connaissent en général plusieurs langages de programmation et non pas un seul. On se concentre rarement sur un seul langage de programmation.

Bien entendu, il faut bien commencer par l'un d'eux. La bonne nouvelle, c'est que vous pouvez commencer par celui que vous voulez ! Les principes des langages sont souvent les mêmes, vous ne serez pas trop dépaysés d'un langage à l'autre.

Néanmoins, voyons plus en détail ce qui caractérise le C++ par rapport aux autres langages de programmation... Et bien oui, c'est un cours de C++ ne l'oubliez pas !

Que vaut le C++ par rapport aux autres langages ?

Le C++ face aux autres langages

Le C++ : langage de haut ou de bas niveau ?

Parmi les centaines de langages de programmation qui existent, certains sont plus populaires que d'autres. Sans aucun doute, le C++ est un langage *très populaire*. Des sites comme langpop.com tiennent à jour un classement des langages les plus couramment utilisés, si cette information vous intéresse. Comme vous pourrez le constater, le C, le Java et le C++ occupent régulièrement le haut du classement.

La question est : faut-il choisir un langage parce qu'il est populaire ? Il existe des langages très intéressants mais peu utilisés. Le souci avec les langages peu utilisés, c'est qu'il est difficile de trouver des gens pour vous aider et vous conseiller quand vous avez un problème. Voilà entre autres pourquoi le C++ est un bon choix pour qui veut débiter : il y a suffisamment de gens qui développent en C++ pour que vous n'ayez pas à craindre de vous retrouver tous seuls !

Bien entendu, il y a d'autres critères que la popularité. Le plus important à mes yeux est le niveau du langage. Il existe des langages de *haut niveau* et d'autres de plus *bas niveau*.



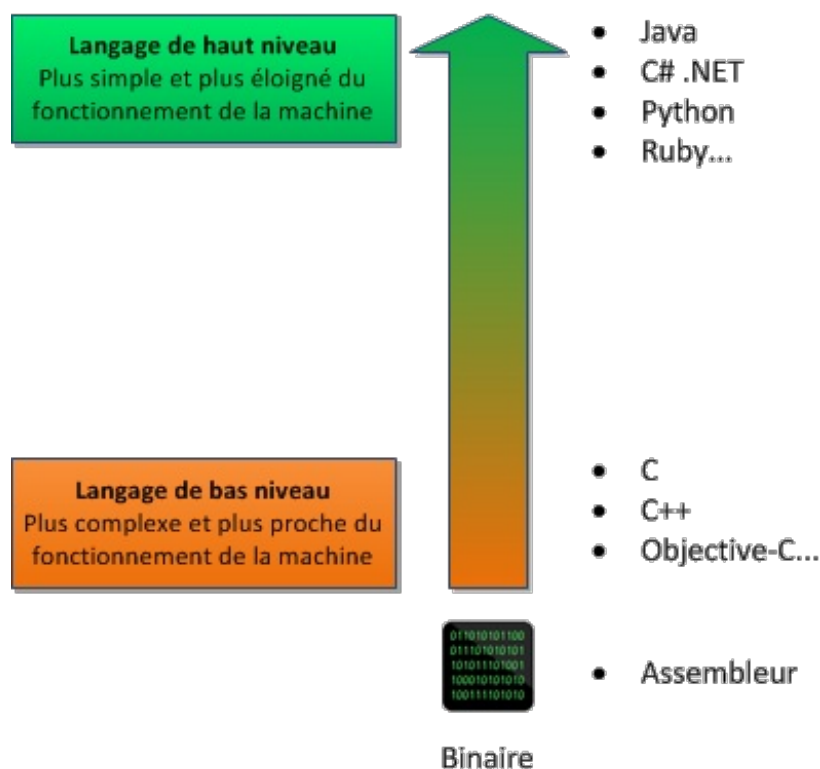
Qu'est-ce qu'un langage de haut niveau ?

C'est un langage assez éloigné du binaire (et donc du fonctionnement de la machine), qui vous permet généralement de développer de façon plus souple et rapide.

Par opposition, un langage de bas niveau est plus proche du fonctionnement de la machine : il demande en général un peu plus d'efforts mais vous donne aussi plus de contrôle sur ce que vous faites. C'est à double tranchant.

Le C++ ? On considère qu'il fait partie de la seconde catégorie : c'est un langage dit « de bas niveau ». Mais que cela ne vous fasse pas peur ! Même si programmer en C++ peut se révéler assez complexe, vous aurez entre les mains un langage très puissant et particulièrement rapide. En effet, si l'immense majorité des jeux sont développés en C++, c'est parce qu'il s'agit du langage qui allie le mieux puissance et rapidité. Voilà ce qui en fait un langage incontournable.

Le schéma ci-dessous représente quelques langages de programmation classés par « niveau » (figure suivante).



Vous constaterez qu'il est en fait possible de programmer en binaire grâce à un langage très basique appelé l'assembleur. Étant donné qu'il faut déployer des efforts surhumains pour coder ne serait-ce qu'une calculatrice, on préfère le plus souvent utiliser un langage de programmation.



En programmation, la notion de « niveau » est relative. Globalement, on peut dire que le C++ est « bas niveau » par rapport au Python, mais il est plus « haut niveau » que l'assembleur. Tout dépend de quel point de vue on se place.

Résumé des forces du C++

- Il est **très répandu**. Comme nous l'avons vu, il fait partie des langages de programmation les plus utilisés sur la planète. On trouve donc beaucoup de documentation sur Internet et on peut facilement avoir de l'aide sur les forums. Il paraît même qu'il y a des gens sympas qui écrivent des cours pour débutants dessus. 🤖
- Il est **rapide**, très rapide même, ce qui en fait un langage de choix pour les applications critiques qui ont besoin de performances. C'est en particulier le cas des jeux vidéo, mais aussi des outils financiers ou de certains programmes militaires qui doivent fonctionner en temps réel.
- Il est **portable** : un même code source peut théoriquement être transformé sans problème en exécutable sous Windows, Mac OS et Linux. Vous n'aurez pas besoin de réécrire votre programme pour d'autres plates-formes !
- Il existe de **nombreuses bibliothèques** pour le C++. Les bibliothèques sont des extensions pour le langage, un peu comme des plug-ins. De base, le C++ ne sait pas faire grand chose mais, en le combinant avec de bonnes bibliothèques, on peut créer des programmes 3D, réseaux, audio, fenêtres, etc.
- Il est **multi-paradigmes** (ouch !). Ce mot barbare signifie qu'on peut programmer de différentes façons en C++. Vous êtes encore un peu trop débutants pour que je vous présente tout de suite ces techniques de programmation mais l'une des plus célèbres est la *Programmation Orientée Objet* (POO). C'est une technique qui permet de simplifier l'organisation du code dans nos programmes et de rendre facilement certains morceaux de codes réutilisables. La partie II de ce cours sera entièrement dédiée à la POO !

Bien entendu, le C++ n'est pas LE langage incontournable. Il a lui-même ses défauts par rapport à d'autres langages, sa complexité en particulier. Vous avez beaucoup de contrôle sur le fonctionnement de votre ordinateur (et sur la gestion de la mémoire) : cela offre une grande puissance mais, si vous l'utilisez mal, vous pouvez plus facilement faire planter votre programme. Ne vous en faites pas, nous découvrirons tout cela progressivement dans ce cours.

Petit aperçu du C++

Pour vous donner une idée, voici un programme très simple affichant le message « Hello world! » à l'écran. « Hello World » est traditionnellement le premier programme que l'on effectue lorsqu'on commence la programmation. Ce sera l'un des premiers codes source que nous étudierons dans les prochains chapitres.

Code : C++

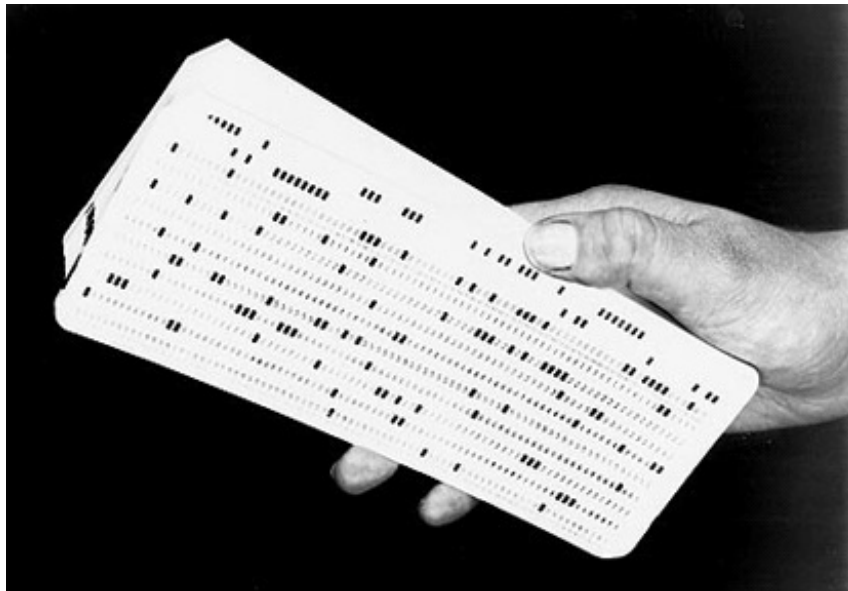
```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

La petite histoire du C++

La programmation a déjà une longue histoire derrière elle. Au début, il n'existait même pas de clavier pour programmer ! On utilisait des cartes perforées comme celle ci-dessous pour donner des instructions à l'ordinateur (figure suivante).



Autant vous dire que c'était long et fastidieux !

De l'Algol au C++

Les choses ont ensuite évolué, heureusement. Le clavier et les premiers langages de programmation sont apparus :

- 1958 : il y a longtemps, à l'époque où les ordinateurs pesaient des tonnes et faisaient la taille de votre maison, on a commencé à inventer un langage de programmation appelé l'**Algol**.
- 1960-1970 : ensuite, les choses évoluant, on a créé un nouveau langage appelé le **CPL**, qui évolua lui-même en **BCPL**, puis qui prit le nom de **langage B** (vous n'êtes pas obligés de retenir tout ça par coeur).
- 1970 : puis, un beau jour, on en est arrivé à créer encore un autre langage qu'on a appelé... le **langage C**. Ce langage, s'il a subi quelques modifications, reste encore un des langages les plus utilisés aujourd'hui. %le plus utilisé, d'après langpop.com cité plus haut !%
- 1983 : un peu plus tard, on a proposé d'ajouter des choses au langage C, de le faire évoluer. Ce nouveau langage, que l'on a appelé « C++ », est entièrement basé sur le C. **Le langage C++** n'est en fait rien d'autre que le langage C avec plusieurs nouveautés. Il s'agit de concepts de programmation poussés comme la programmation orientée objet, le polymorphisme, les flux... Bref, des choses bien compliquées pour nous pour le moment mais dont nous aurons l'occasion de reparler par la suite !



Une minute... Si le C++ est en fait une amélioration du C, pourquoi y a-t-il encore tant de gens qui développent en C ?

Tout le monde n'a pas besoin des améliorations apportées par le langage C++. Le C est à lui seul suffisamment puissant pour être à la base des systèmes d'exploitation comme Linux, Mac OS X et Windows.

Ceux qui n'ont pas besoin des améliorations (mais aussi de la complexité !) apportées par le langage C++ se contentent donc très bien du langage C et ce, malgré son âge. Comme quoi, un langage peut être vieux et rester d'actualité.

Le concepteur

C'est Bjarne Stroustrup, un informaticien originaire du Danemark, qui a conçu le langage C++. Insatisfait des possibilités offertes par le C, il a créé en 1983 le C++ en y ajoutant les possibilités qui, selon lui, manquaient.

Bjarne Stroustrup est aujourd'hui professeur d'informatique à l'université Texas A&M, aux Etats-Unis. Il s'agit d'une importante figure de l'univers informatique qu'il faut connaître, au moins de nom (du moins si vous arrivez à le retenir !).

De nombreux langages de programmation se sont par la suite inspirés du C++. C'est notamment le cas du langage Java.

Le langage C++, bien que relativement ancien, continue à être amélioré. Une nouvelle version, appelée « C++1x », est d'ailleurs en cours de préparation. Il ne s'agit pas d'un nouveau langage mais d'une mise à jour du C++. Les nouveautés qu'elle apporte sont cependant trop complexes pour nous, nous n'en parlerons donc pas ici !

En résumé

- Les programmes permettent de réaliser toutes sortes d'actions sur un ordinateur : navigation sur le Web, rédaction de textes, manipulation des fichiers, etc.
- Pour réaliser des programmes, on écrit des instructions pour l'ordinateur dans un langage de programmation. C'est le code source.
- Le code doit être traduit en binaire par un outil appelé compilateur pour qu'il soit possible de lancer le programme. L'ordinateur ne comprend en effet que le binaire.
- Le C++ est un langage de programmation très répandu et rapide. C'est une évolution du langage C car il offre en particulier la possibilité de programmer en orienté objet, une technique de programmation puissante qui sera présentée dans ce livre.

Les logiciels nécessaires pour programmer

Maintenant que l'on en sait un peu plus sur le C++, si on commençait à pratiquer ?

Ah mais oui, c'est vrai : vous ne pouvez pas programmer tant que vous n'avez pas installé les bons logiciels ! En effet, il faut installer certains logiciels spécifiques pour programmer en C++. Dans ce chapitre, nous allons les installer et les découvrir ensemble.

Un peu de patience : dès le prochain chapitre nous pourrons enfin commencer à véritablement programmer ! 😊

Les outils nécessaires au programmeur

Alors à votre avis, de quels outils un programmeur a-t-il besoin ?

Si vous avez attentivement suivi le chapitre précédent, vous devez en connaître au moins un !

Vous voyez de quoi je parle ?

?

?

?

Vraiment pas ? 😊

Eh oui, il s'agit du **compilateur**, ce fameux programme qui permet de traduire votre langage C++ en langage binaire !

Comme je vous l'avais un peu déjà dit dans le premier chapitre, il existe plusieurs compilateurs pour le langage C++. Nous allons voir que le choix du compilateur ne sera pas très compliqué dans notre cas.

Bon, de quoi d'autre a-t-on besoin ?

Je ne vais pas vous laisser deviner plus longtemps. Voici le strict minimum pour un programmeur :

- **Un éditeur de texte** pour écrire le code source du programme en C++. En théorie un logiciel comme le Bloc-Notes sous Windows, ou "vi" sous Linux fait l'affaire. L'idéal, c'est d'avoir un éditeur de texte intelligent qui colore tout seul le code, ce qui vous permet de vous repérer dedans bien plus facilement. Voilà pourquoi aucun programmeur sain d'esprit n'utilise Bloc-Notes. 😊
- **Un compilateur** pour transformer ("compiler") votre source en binaire.
- **Un débogueur** pour vous aider à traquer les erreurs dans votre programme (on n'a malheureusement pas encore inventé le "correcteur", un truc qui corrigerait tout seul nos erreurs 😊)

A priori, si vous êtes un casse-cou de l'extrême, vous pourriez vous passer de débogueur. Mais bon, je sais pertinemment que dans moins de 5 minutes vous reviendrez en pleurnichant me demander où on peut trouver un débogueur qui marche bien. 😊

A partir de maintenant on a 2 possibilités :

- Soit on récupère chacun de ces 3 programmes **séparément**. C'est la méthode la plus compliquée, mais elle fonctionne. Sous Linux en particulier, bon nombre de programmeurs préfèrent utiliser ces 3 programmes séparément. Je ne détaillerai pas cette méthode ici, je vais plutôt vous parler de la méthode simple.
- Soit on utilise un programme "3-en-1" (comme les liquides vaisselle, oui oui) qui combine éditeur de texte, compilateur et débogueur. Ces programmes "3-en-1" sont appelés **IDE** (ou en français "EDI" pour "Environnement de développement intégré").

Il existe plusieurs environnements de développement. Vous aurez peut-être un peu de mal à choisir celui qui vous plaît au début. Une chose est sûre en tout cas : vous pouvez faire n'importe quel type de programme, quel que soit l'IDE que vous choisissez.

Les projets

Quand vous réalisez un programme, on dit que vous travaillez sur un **projet**. Un projet est constitué de plusieurs fichiers de code source : des fichiers .cpp, .h, les images du programme, etc.

Le rôle d'un IDE est de rassembler tous ces fichiers d'un projet au sein d'une même interface. Comme ça, vous avez accès à tous les éléments de votre programme à portée de clic.

Voilà pourquoi, quand vous voudrez créer un nouveau programme, il faudra demander à l'IDE de vous préparer un "nouveau projet".

Choisissez votre IDE

Il m'a semblé intéressant de vous montrer quelques IDE parmi les plus connus. Tous sont disponibles gratuitement. Personnellement, je navigue un peu entre tous ceux-là et j'utilise l'IDE qui me plaît selon l'humeur du jour. 😊

- Un des IDE que je préfère s'appelle **Code::Blocks**. Il est gratuit et fonctionne sur la plupart des systèmes d'exploitation. Je conseille d'utiliser celui-ci pour débiter (et même pour la suite s'il vous plaît bien !).
Fonctionne sous Windows, Mac et Linux.
- Le plus célèbre IDE sous Windows, c'est celui de Microsoft : **Visual C++**. Il existe à la base en version payante (chère !), mais heureusement il existe une version gratuite intitulée **Visual C++ Express** qui est vraiment très bien (il y a peu de différences avec la version payante). Il est très complet et possède un puissant module de correction des erreurs (debuggage).
Fonctionne sous Windows uniquement.
- Sur Mac OS X, vous pouvez aussi utiliser XCode, généralement fourni sur le CD d'installation de Mac OS X. C'est un IDE très apprécié par tous ceux qui font de la programmation sur Mac.
Fonctionne sous Mac OS X uniquement.



Note pour les utilisateurs de Linux : il existe de nombreux IDE sous Linux, mais les programmeurs expérimentés préfèrent parfois se passer d'IDE et compiler "à la main", ce qui est un peu plus difficile. Vous aurez le temps d'apprendre à faire cela plus tard. En ce qui nous concerne nous allons commencer par utiliser un IDE. Je vous conseille d'installer Code::Blocks si vous êtes sous Linux pour suivre mes explications. Vous pouvez aussi jeter un oeil du côté d'Eclipse pour les développeurs C/C++, un IDE très puissant qui ne sert pas qu'à programmer en Java contrairement à l'idée répandue !



Quel est le meilleur de tous ces IDE ?

Tous ces IDE vous permettront de programmer et de suivre le reste de ce cours sans problème. Certains sont plus complets au niveau des options, d'autres un peu plus intuitifs à utiliser, mais dans tous les cas les programmes que vous créerez seront les mêmes quel que soit l'IDE que vous utilisez. Ce choix n'est donc pas si crucial qu'on pourrait le croire.

Durant tout ce cours, j'utiliserai Code::Blocks. Si vous voulez avoir exactement les mêmes écrans que moi, surtout pour ne pas être perdu au début, je vous recommande donc de commencer par installer Code::Blocks.

Code::Blocks (Windows, Mac OS, Linux)

Code::Blocks est un IDE libre et gratuit, disponible **pour Windows, Mac et Linux**.

Code::Blocks n'est disponible pour le moment qu'en anglais. Ca ne devrait PAS vous repousser à l'utiliser. Nous utiliserons très peu les menus en fait. 😊

Sachez toutefois que quand vous programmerez vous serez de toute façon confronté bien souvent à des documentations en anglais. Voilà donc une raison de plus pour s'entraîner à utiliser cette langue.

Télécharger Code::Blocks

Rendez-vous [sur la page de téléchargements de Code::Blocks](#).

- Si vous êtes sous Windows, repérez la section "Windows" un peu plus bas sur cette page. Téléchargez le logiciel en prenant le programme qui contient mingw dans le nom (ex. : codeblocks-10.05mingw-setup.exe). L'autre version étant sans compilateur, vous auriez eu du mal à compiler vos programmes. 🤖
- Si vous êtes sous Linux, le mieux est encore d'installer Code::Blocks via les dépôts (avec la commande `apt-get` sous Ubuntu par exemple). Il vous faudra aussi installer le compilateur à part : c'est le paquet build-essential. Vous devriez donc rentrer cette commande pour installer le compilateur et l'IDE Code::Blocks :

Code : Console


```
apt-get install build-essential codeblocks
```

- Enfin, sous Mac, choisissez le fichier le plus récent de la liste (ex : codeblocks-10.05-p2-mac.zip).



J'insiste là-dessus : si vous êtes sous Windows, **téléchargez la version incluant mingw** dans le nom du programme d'installation. Si vous prenez la mauvaise version, vous ne pourrez pas compiler vos programmes par la suite !

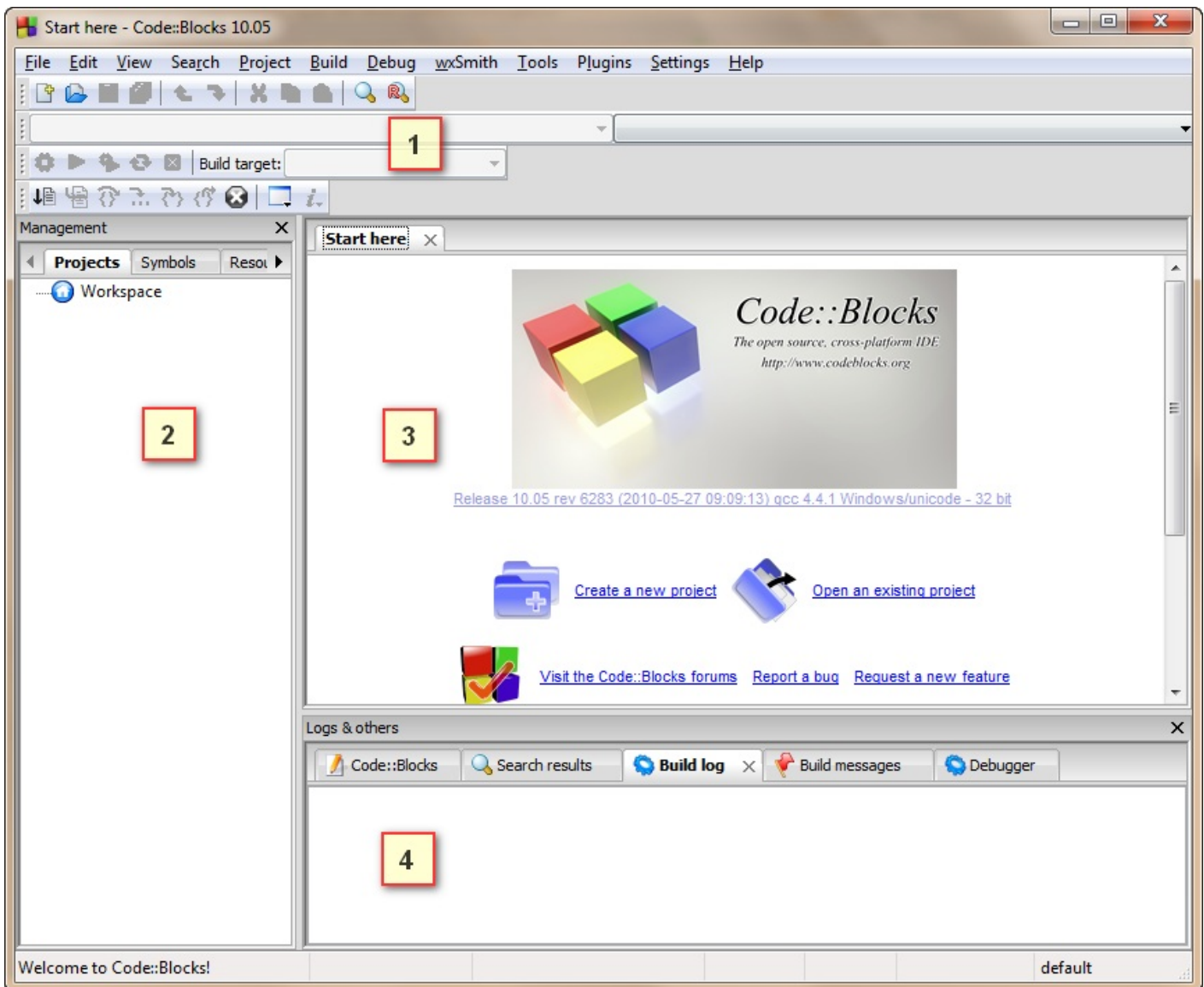


Windows 2000 / XP / Vista / 7:

File	Date	Size	Download from
codeblocks-10.05-setup.exe	27 May 2010	23.3 MB	BerliOS
codeblocks-10.05mingw-setup.exe	27 May 2010	74.0 MB	BerliOS

NOTE: The codeblocks-10.05mingw-setup.exe file *includes* the GCC compiler and GDB debugger from **MinGW**.

L'installation est très simple et rapide. Laissez toutes les options par défaut et lancez le programme.



On distingue 4 grandes sections dans la fenêtre, numérotées sur l'image :

1. **La barre d'outils** : elle comprend de nombreux boutons, mais seuls quelques-uns d'entre eux nous seront régulièrement utiles. J'y reviendrai plus loin.
2. **La liste des fichiers du projet** : c'est à gauche que s'affiche la liste de tous les fichiers source de votre programme. Notez que sur cette capture aucun projet n'a été créé donc on ne voit pas encore de fichiers à l'intérieur de la liste. Vous verrez cette section se remplir dans cinq minutes en lisant la suite du cours.
3. **La zone principale** : c'est là que vous pourrez écrire votre code en langage C++ !
4. **La zone de notification** : aussi appelée la "Zone de la mort", c'est ici que vous verrez les erreurs de compilation s'afficher si votre code comporte des erreurs. Cela arrive très régulièrement !

Intéressons-nous maintenant à une section particulière de la barre d'outils. Vous trouverez les boutons suivants (dans l'ordre) "Compiler", "Exécuter", "Compiler & Exécuter" et "Tout recompiler". Retenez-les, nous les utiliserons régulièrement.



- **Compiler** : tous les fichiers source de votre projet sont envoyés au compilateur qui va se charger de créer un exécutable. S'il y a des erreurs (ce qui a de fortes chances d'arriver 😞), l'exécutable ne sera pas créé et on vous indiquera les erreurs en bas de Code::Blocks.
- **Exécuter** : cette icône lance juste le dernier exécutable que vous avez compilé. Cela vous permettra donc de tester votre programme et voir ainsi ce qu'il donne. Dans l'ordre, si vous avez bien suivi, on doit d'abord compiler, puis exécuter pour tester ce que ça donne. On peut aussi utiliser le 3ème bouton...
- **Compiler & Exécuter** : pas besoin d'être un génie pour comprendre que c'est la combinaison des 2 boutons précédents.

C'est d'ailleurs ce bouton que vous utiliserez le plus souvent. Notez que s'il y a des erreurs pendant la compilation (pendant la génération de l'exécutable), le programme ne sera pas exécuté. A la place, vous aurez droit à une belle liste d'erreurs à corriger. 😞

- **Tout reconstruire** : quand vous faites " Compiler ", Code::Blocks ne recompile en fait que les fichiers que vous avez modifiés et pas les autres. Parfois, je dis bien parfois, vous aurez besoin de demander à Code::Blocks de vous recompiler tous les fichiers. On verra plus tard quand on a besoin de ce bouton, et vous verrez plus en détail le fonctionnement de la compilation dans un chapitre futur. Pour l'instant, on se contente de savoir le minimum nécessaire pour pas tout mélanger.

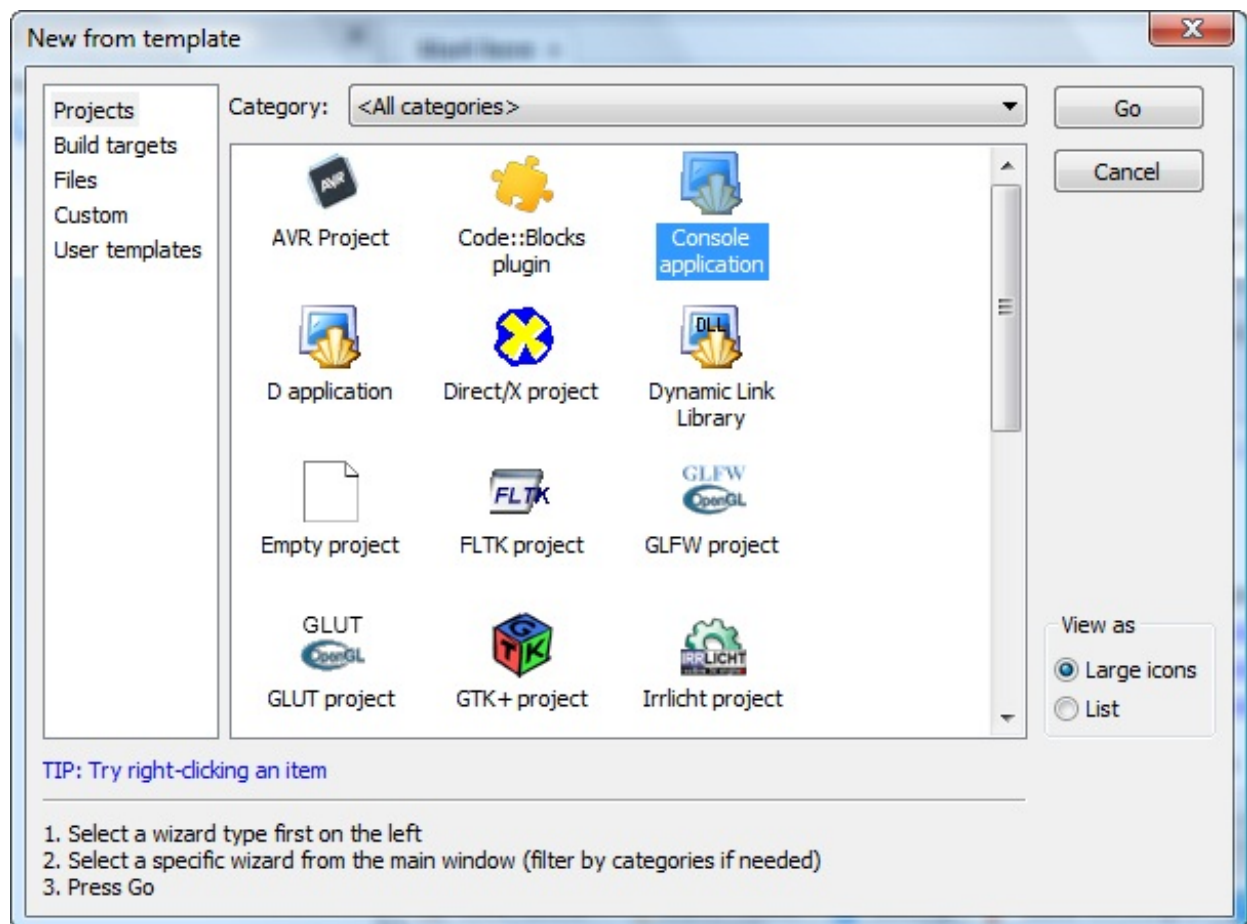
Ce bouton ne nous sera donc pas utile de suite.



Je vous conseille d'utiliser les raccourcis plutôt que de cliquer sur les boutons, parce que c'est quelque chose qu'on fait vraiment très très souvent. Retenez en particulier qu'il faut taper sur F9 pour faire " Compiler & Exécuter ".

Créer un nouveau projet

Pour créer un nouveau projet c'est très simple : allez dans le menu File / New / Project.
Dans la fenêtre qui s'ouvre, choisissez "Console application" :



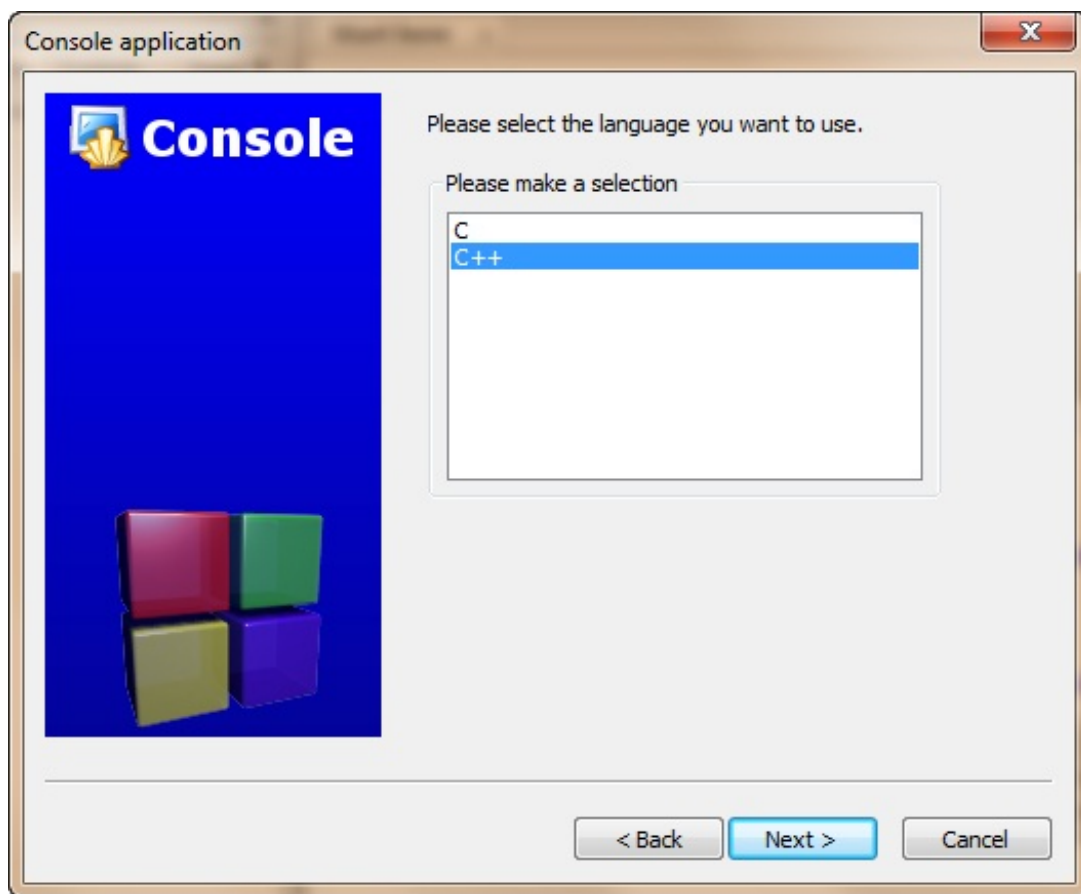
Comme vous pouvez le voir, Code::Blocks propose de réaliser pas mal de types de programmes différents qui utilisent des bibliothèques connues comme la SDL (2D), OpenGL (3D), Qt et wxWidgets (Fenêtres) etc etc... Pour l'instant, ces icônes servent plutôt à faire joli car les bibliothèques ne sont pas installées sur votre ordinateur, vous ne pourrez donc pas les faire marcher.

Nous nous intéresserons à ces autres types de programmes bien plus tard. En attendant il faudra vous contenter de "Console", car vous n'avez pas encore le niveau nécessaire pour créer les autres types de programmes.

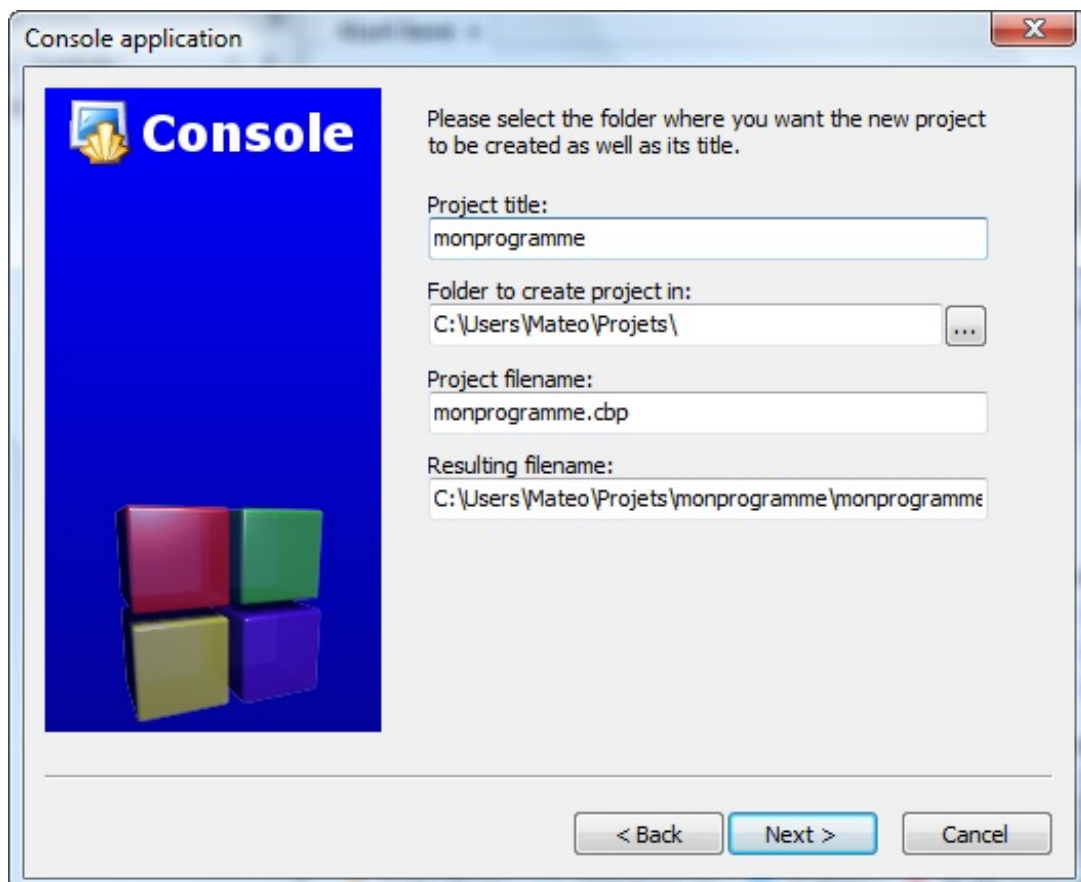
Cliquez sur "Go" pour créer le projet. Un assistant s'ouvre.

Faites "Next", la première page ne servant à rien.

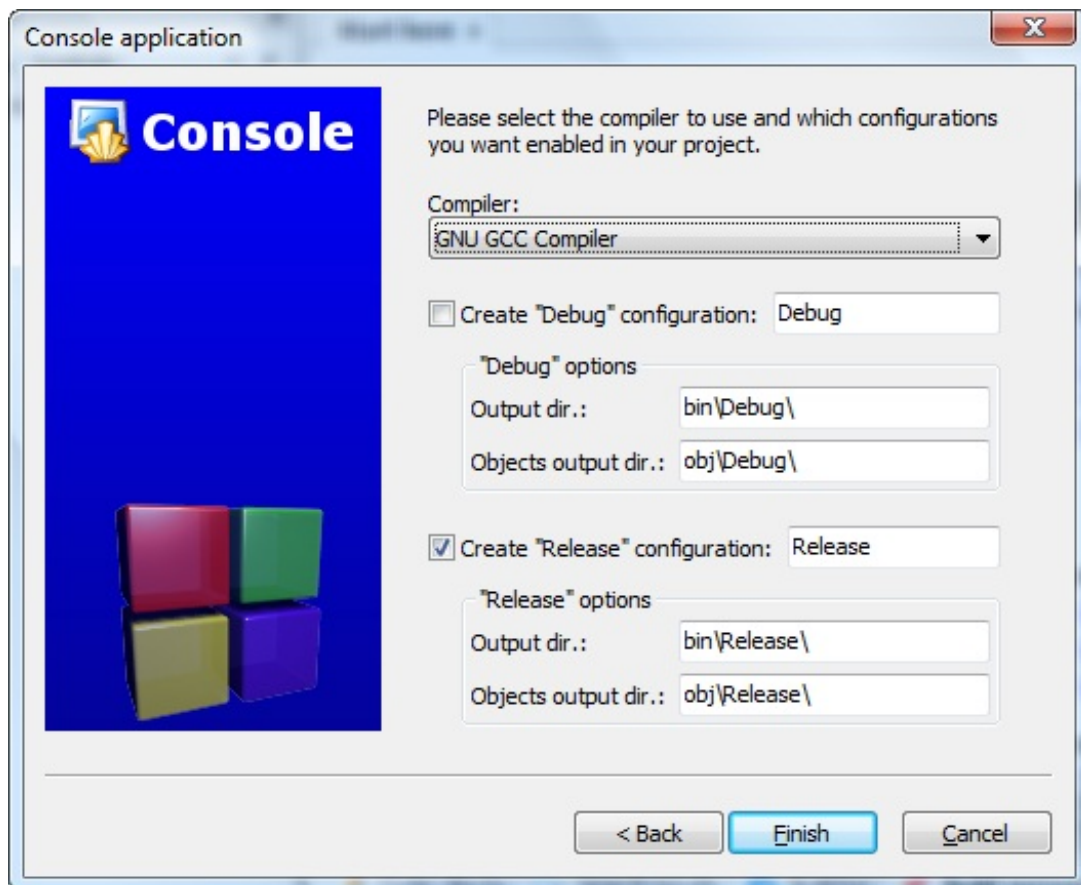
On vous demande ensuite si vous allez faire du C ou du C++ : répondez C++.



On vous demande le nom de votre projet, et dans quel dossier les fichiers source seront enregistrés :



Enfin, la dernière page vous permet de choisir de quelle façon le programme doit être compilé. Vous pouvez laisser les options par défaut, ça n'aura pas d'incidence pour ce que nous allons faire dans l'immédiat (veillez à ce que "Debug" ou "Release" au moins soit coché).



Cliquez sur "Finish", c'est bon !

Code::Blocks vous créera un premier projet avec déjà un tout petit peu de code source dedans. 😊

Dans le cadre de gauche "Projects", développez l'arborescence en cliquant sur le petit "+" pour afficher la liste des fichiers du projet. Vous devriez avoir au moins un main.cpp que vous pourrez ouvrir en double-cliquant dessus.

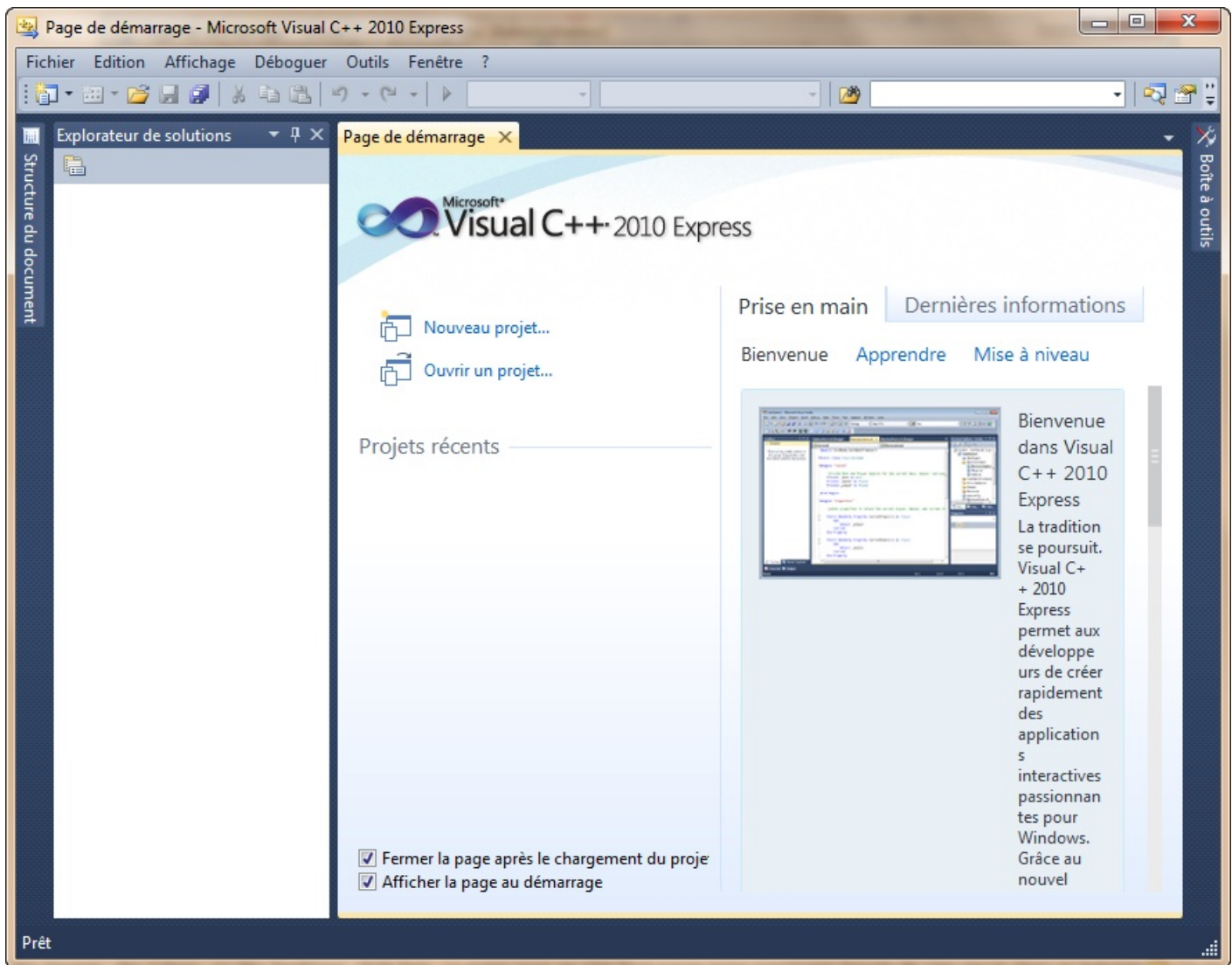
Et voilà !

Visual C++ (Windows seulement)

Quelques petits rappels sur Visual C++ :

- C'est l'IDE de Microsoft
- Il est à la base payant, mais Microsoft a sorti une version gratuite intitulée Visual C++ Express.

Nous allons bien entendu voir ici la version gratuite, Visual C++ Express. 😊

*Aperçu de Visual C++ Express*

Quelles sont les différences avec le "vrai" Visual ?

Il n'y a pas d'éditeur de ressources (vous permettant de dessiner des images, des icônes, ou des fenêtres). Mais bon, ça entre nous on s'en fout parce qu'on n'aura pas besoin de s'en servir dans ce tutoriel. 😊 Ce ne sont pas des fonctionnalités indispensables bien au contraire.

Vous trouverez les instructions pour télécharger Visual C++ Express à cette adresse :

[Site de Visual C++ Express Edition](#)

Sélectionnez Visual C++ Express Français un peu plus bas sur la page.

Visual C++ Express est en français et est totalement gratuit. Ce n'est donc pas une version d'essai limitée dans le temps.

C'est une chance d'avoir un IDE aussi puissant que celui de Microsoft disponible gratuitement, donc ne la laissez pas passer. 😊

Installation

L'installation devrait normalement se passer sans encombre. Le programme d'installation va télécharger la dernière version de

www.siteduzero.com

Visual sur Internet.

Je vous conseille de laisser les options par défaut.

A la fin, on vous dit qu'il faut vous enregistrer dans les 30 jours. Pas de panique, c'est gratuit et rapide mais il faut le faire. Cliquez sur le lien qui vous est donné : vous arrivez sur le site de Microsoft. Connectez-vous avec votre compte Windows Live ID (équivalent du compte hotmail ou msn) ou créez-en un si vous n'en avez pas, puis répondez au petit questionnaire.

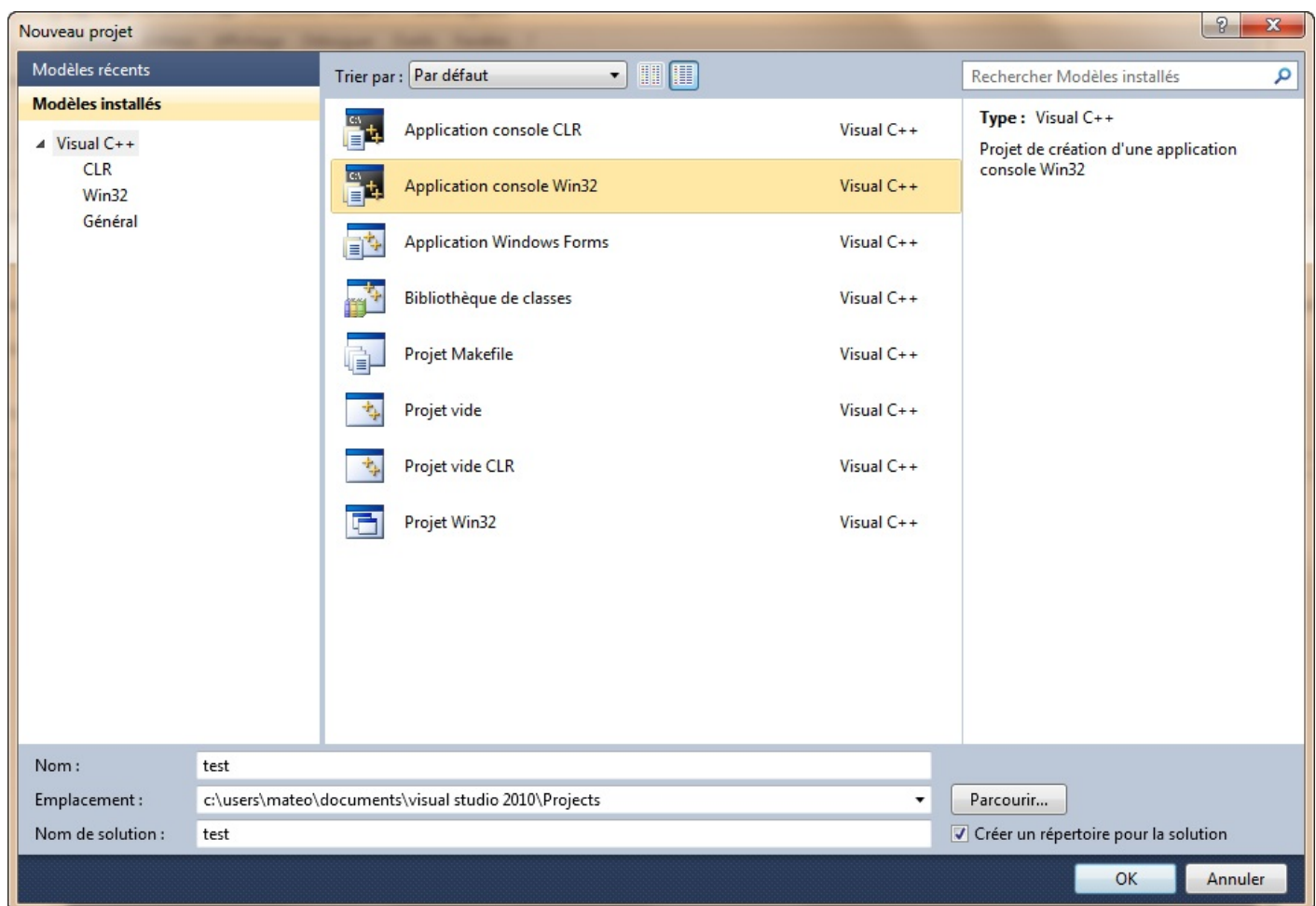
On vous donnera à la fin une clé d'enregistrement. Vous devrez recopier cette clé dans le menu "?" / "Inscrire le produit".

Créer un nouveau projet

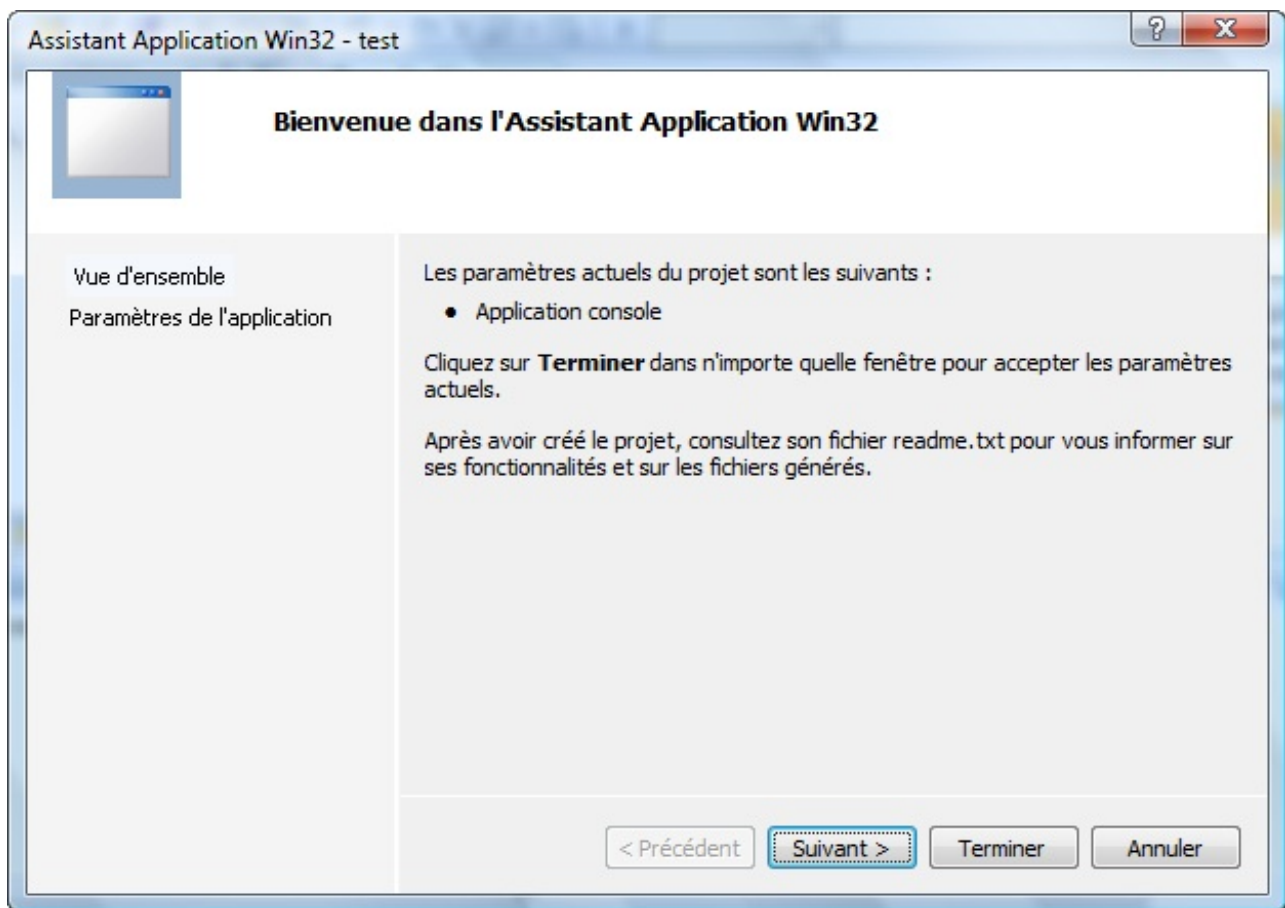
Pour créer un nouveau projet sous Visual, allez dans le menu Fichier / Nouveau / Projet.

Sélectionnez "Win32" dans la colonne de gauche, puis "Application console Win32" à droite.

Entrez un nom pour votre projet, par exemple "test" :

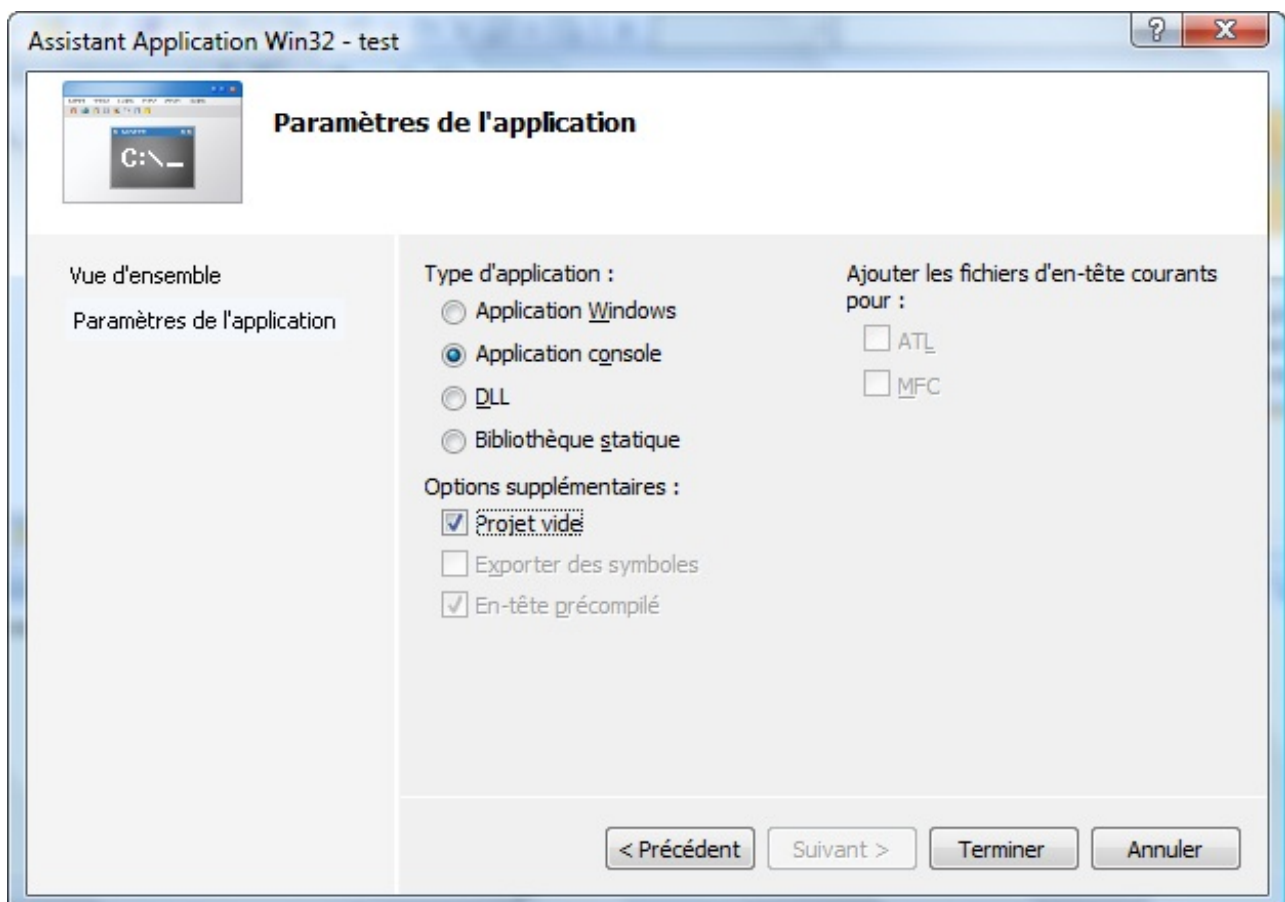


Validez. Une nouvelle fenêtre s'ouvre :



Cette fenêtre ne sert à rien. 😊

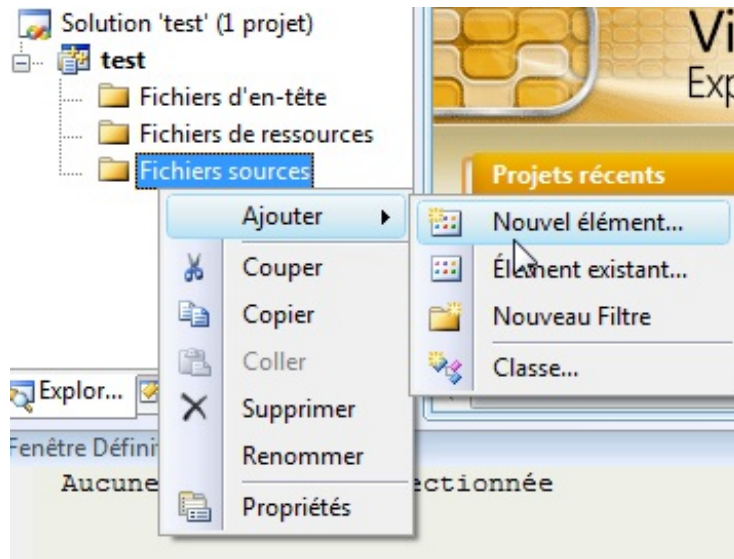
Par contre, cliquez sur "Paramètres de l'application" dans la colonne de gauche :



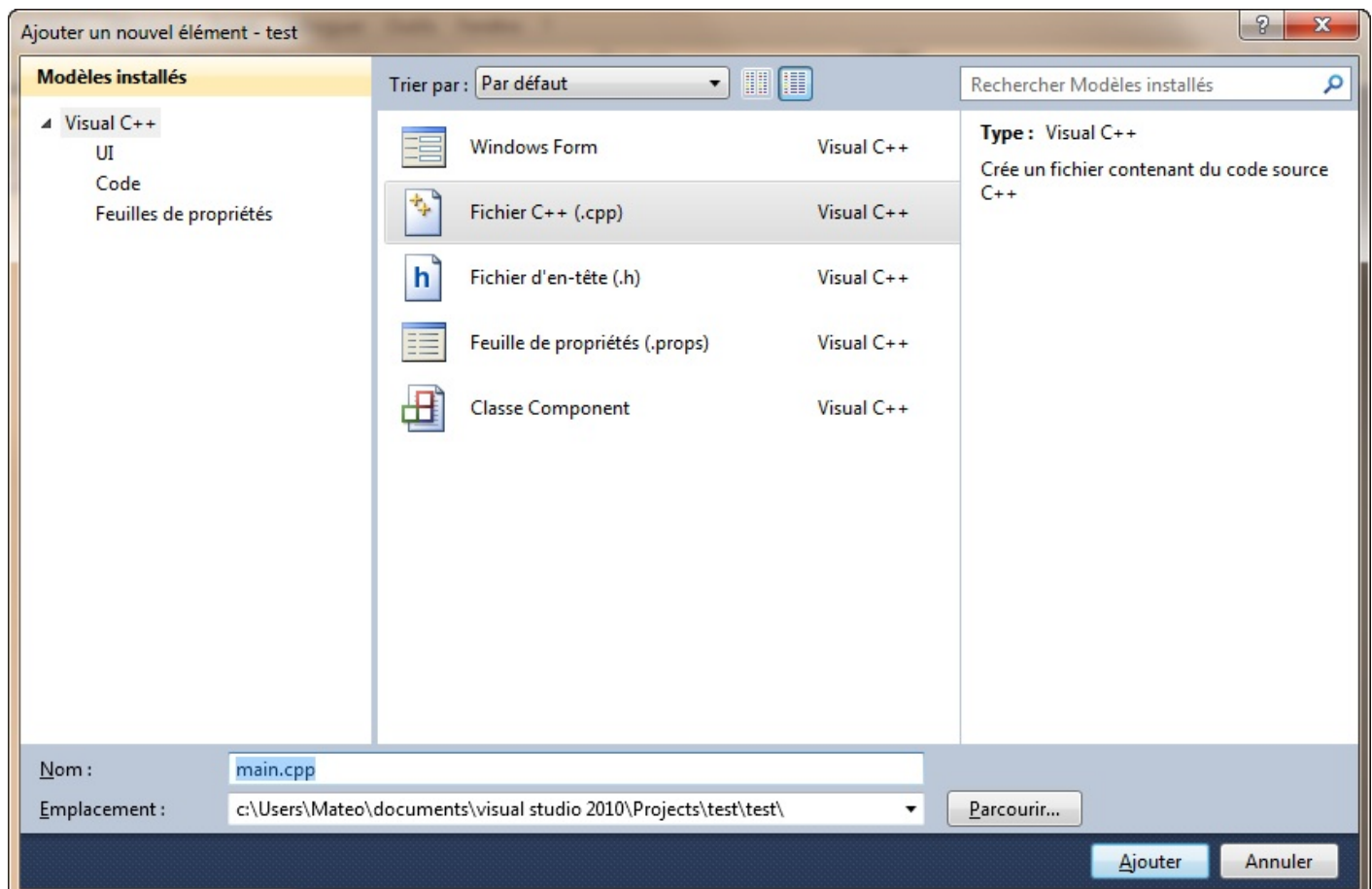
Veillez à ce que "Projet vide" soit coché comme sur ma capture d'écran.
Puis, cliquez sur "Terminer".

Ajouter un nouveau fichier source

Votre projet est pour l'instant bien vide. Faites un clic droit sur le dossier "Fichiers sources" situé sur votre gauche, puis allez dans Ajouter / Nouvel élément :



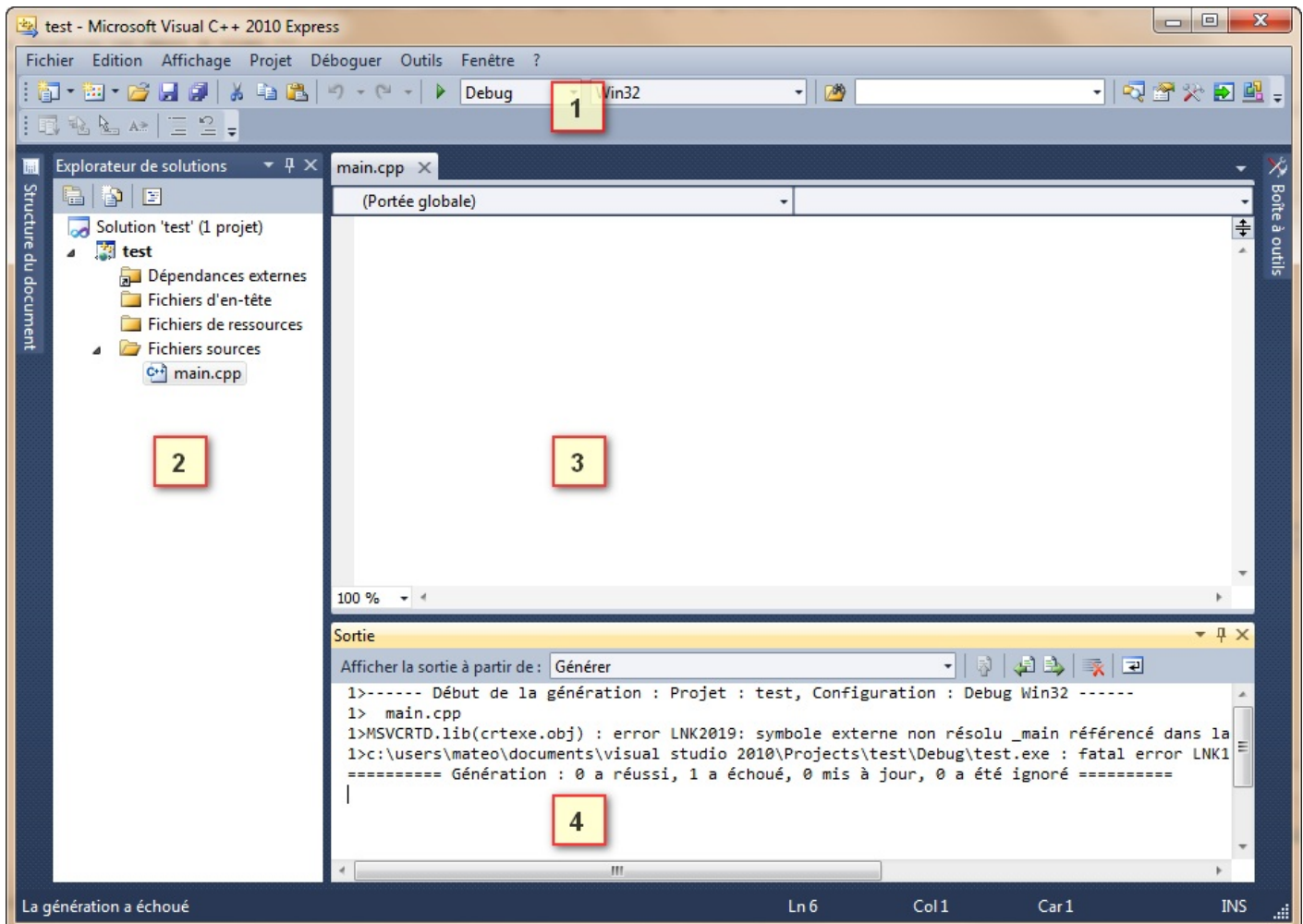
Une fenêtre s'ouvre.
Sélectionnez "Fichier C++ (.cpp)". Entrez le nom "main.cpp" pour votre fichier :



Cliquez sur "Ajouter". C'est bon, vous allez pouvoir commencer à écrire du code !

La fenêtre principale de Visual

Voyons ensemble le contenu de la fenêtre principale de Visual C++ Express :



On va rapidement (re)voir quand même ce que signifient chacune des parties :

1. La barre d'outils, tout ce qu'il y a de plus standard. Ouvrir, enregistrer, enregistrer tout, couper, copier, coller etc. Par défaut, il semble qu'il n'y ait pas de bouton de barre d'outils pour compiler. Vous pouvez les rajouter en faisant un clic droit sur la barre d'outils, puis en choisissant "Débuguer" et "Générer" dans la liste. Toutes ces icônes de compilation ont leur équivalent dans les menus "Générer" et "Débuguer". Si vous faites "Générer", cela créera l'exécutable (ça signifie "Compiler" pour Visual). Si vous faites "Débuguer / Exécuter", on devrait vous proposer de compiler avant d'exécuter le programme. F7 permet de générer le projet, et F5 de l'exécuter.
2. Dans cette zone très importante vous voyez normalement la liste des fichiers de votre projet. Cliquez sur l'onglet "Explorateur de solutions" en bas si ce n'est déjà fait. Vous devriez voir que Visual crée déjà des dossiers pour séparer les différents types de fichiers de votre projet (sources, en-têtes et ressources). Nous verrons un peu plus tard quels sont les différents types de fichiers qui constituent un projet. 😊
3. La partie principale. C'est là qu'on modifie les fichiers source.
4. C'est là encore la "zone de la mort", celle où on voit apparaître toutes les erreurs de compilation. C'est dans le bas de l'écran aussi que Visual affiche les informations de débogage quand vous essayez de corriger un programme buggé. Je vous ai d'ailleurs dit tout à l'heure que j'aimais beaucoup le débogger de Visual, et je pense que je ne suis pas le seul. 😊

Voilà, on a fait le tour de Visual C++.

Vous pouvez aller jeter un œil dans les options (Outils / Options) si ça vous chante, mais n'y passez pas 3 heures. Il faut dire qu'il y a tellement de cases à cocher de partout qu'on ne sait plus trop où donner de la tête. 😊

Xcode (Mac OS seulement)

Il existe plusieurs IDE compatibles Mac. Il y a Code::Blocks bien sûr, mais ce n'est pas le seul.

Je vais vous présenter ici l'IDE le plus célèbre sous Mac : Xcode.



Merci à [prs513rosewood](#) pour les captures d'écran et ses judicieux conseils pour réaliser cette section.

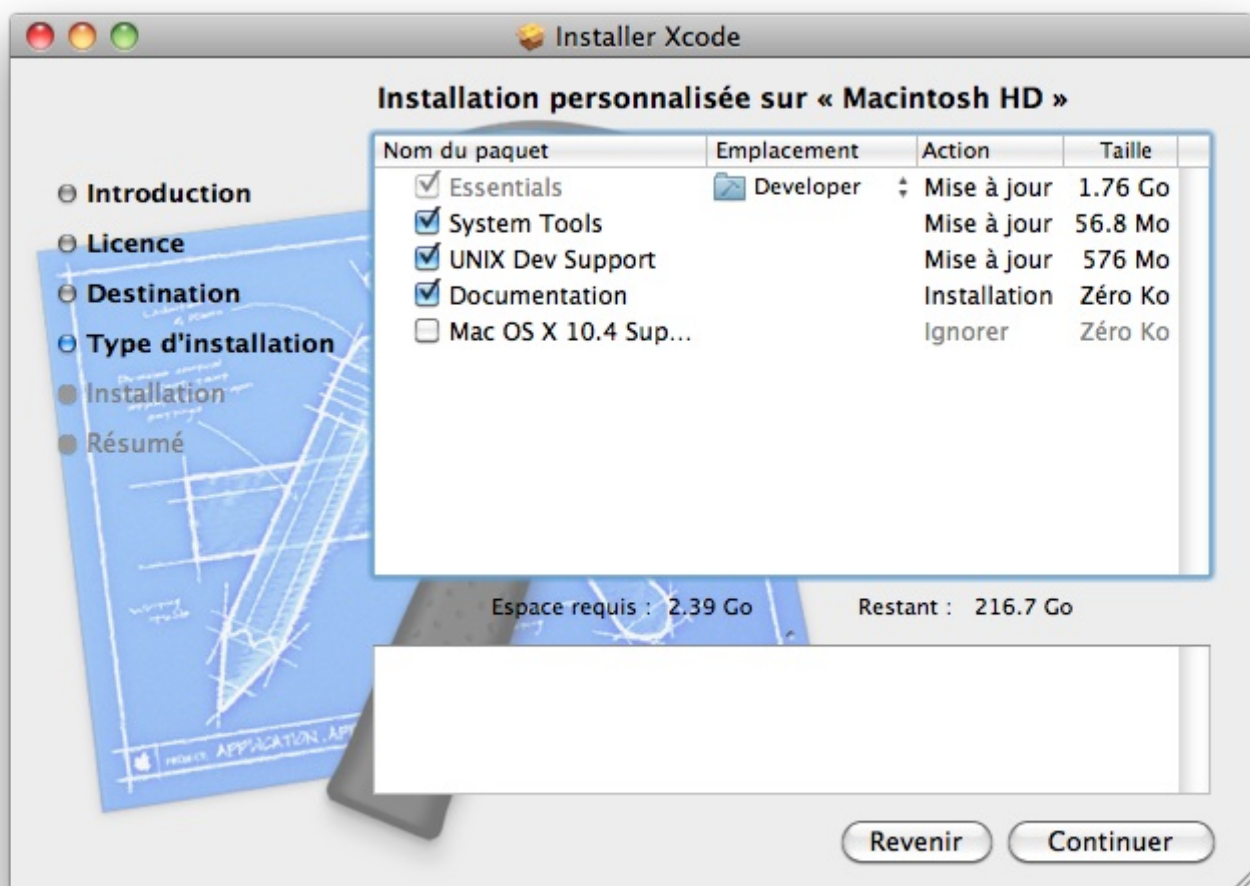
Xcode, où es-tu ?



Tous les utilisateurs de Mac OS ne sont pas des programmeurs. Apple l'a bien compris et n'installe pas par défaut d'IDE avec Mac OS.

Heureusement, pour ceux qui voudraient programmer, tout est prévu. En effet, Xcode est présent sur le CD d'installation de Mac OS.

Insérez donc le CD dans le lecteur. Pour installer Xcode, il faut ouvrir le paquet "Xcode Tools" dans le répertoire "Installation facultative" du disque d'installation. L'installateur démarre :



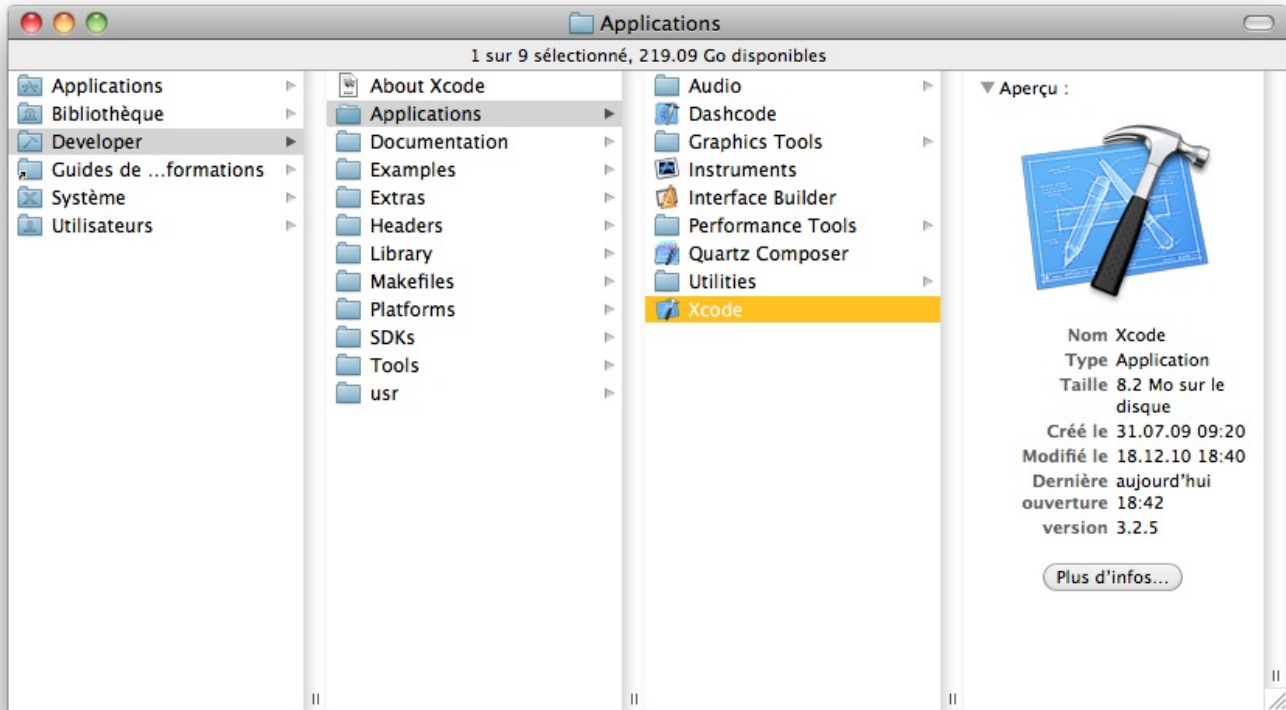
Par ailleurs, je vous conseille de mettre en favoris la [page dédiée aux développeurs](#) sur le site d'Apple. Vous y trouverez une foule d'informations utiles pour le développement sous Mac. Vous pourrez notamment y télécharger plusieurs logiciels pour développer.

N'hésitez pas à vous inscrire à l'ADC (Apple Development Connection), c'est gratuit et vous serez ainsi tenu au courant des

nouveautés.

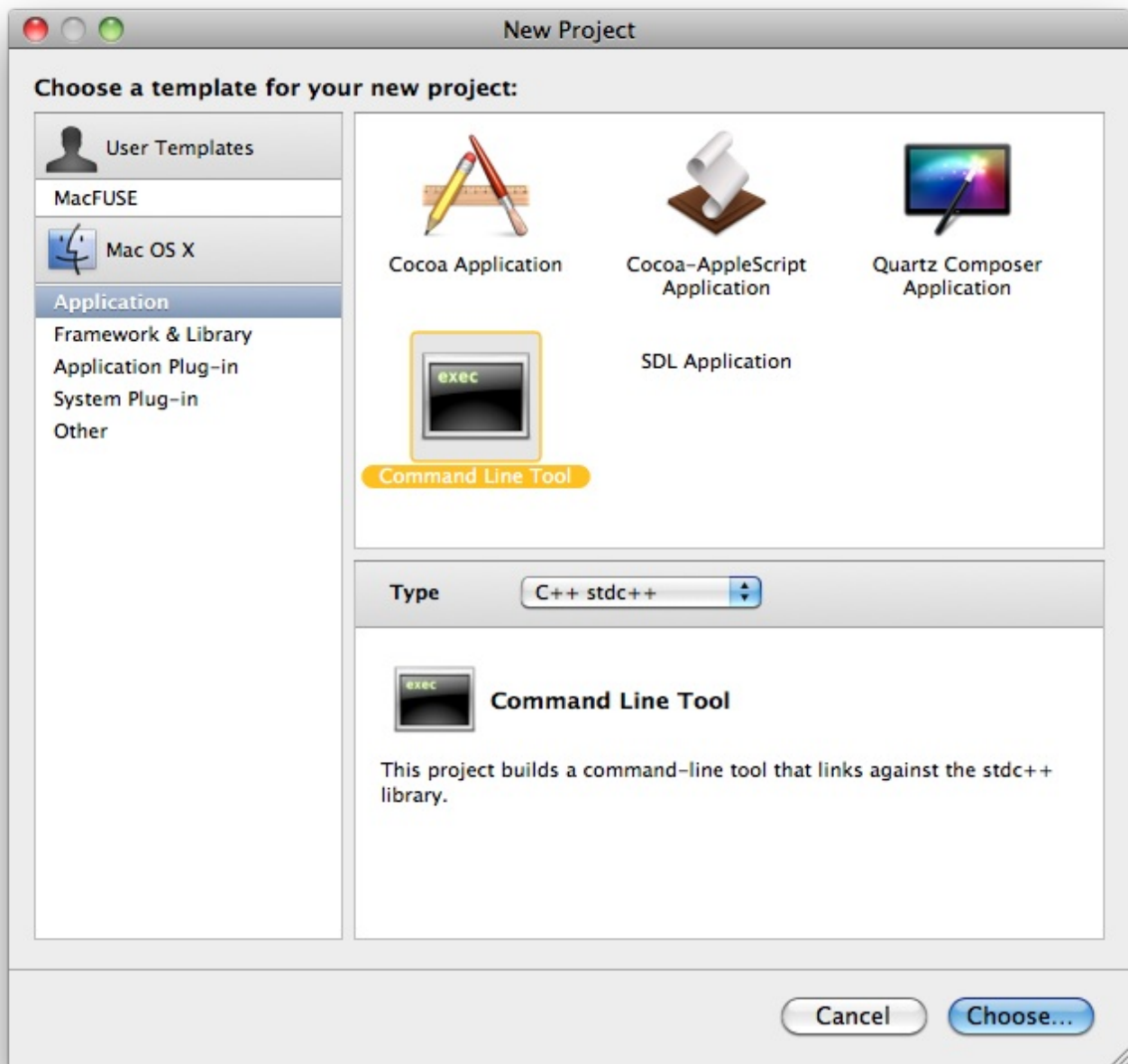
Lancement

Une fois l'installation terminée, l'application Xcode se trouve dans le répertoire /Developer/Applications/ :

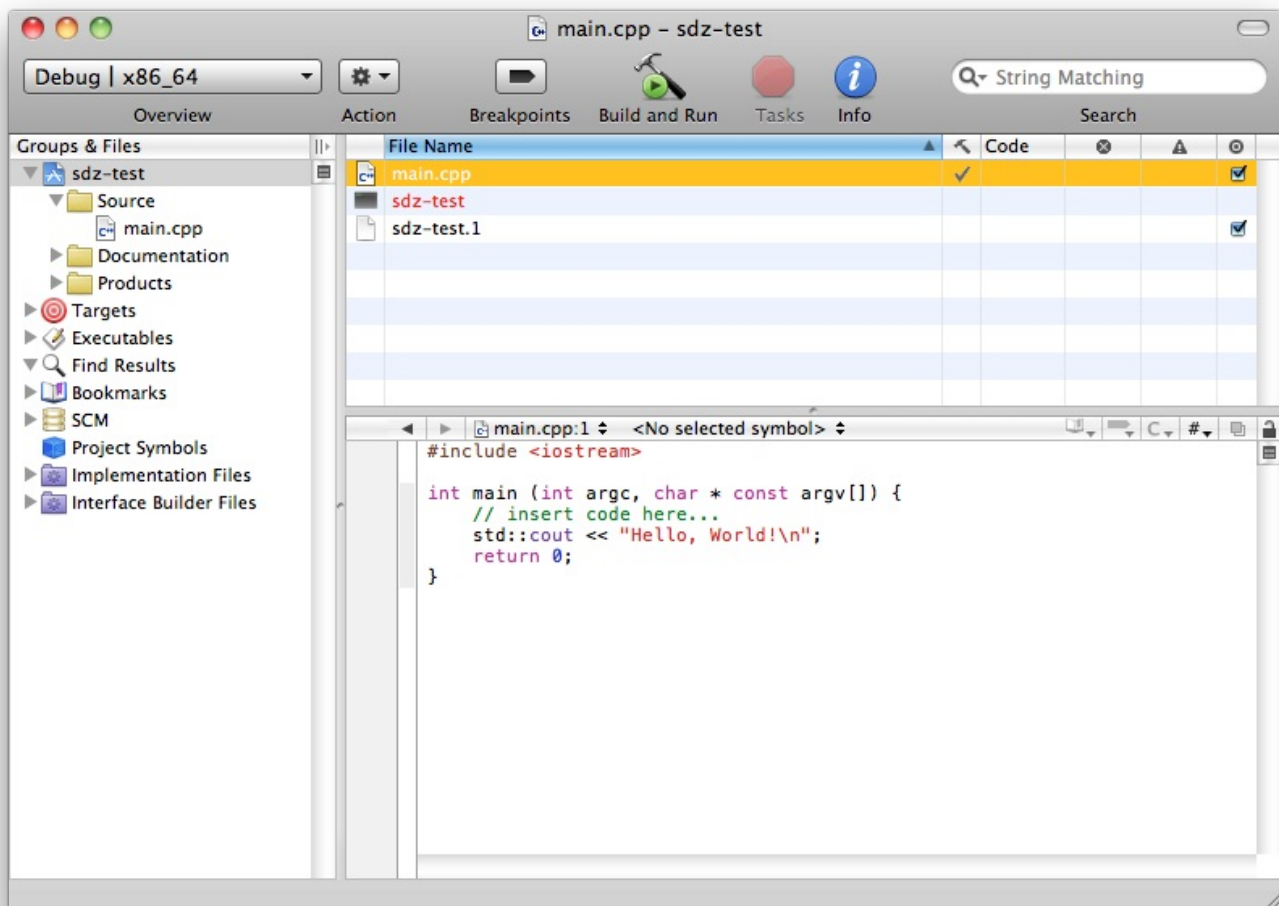


Nouveau projet

Pour créer un nouveau projet, on clique sur "Create a new Xcode project", ou "File > New Project". Il faut choisir le type "Command Line Tool", et sélectionner "C++ sdte++" dans le petit menu déroulant.



Une fois le projet créé, la fenêtre principale de Xcode apparaît :

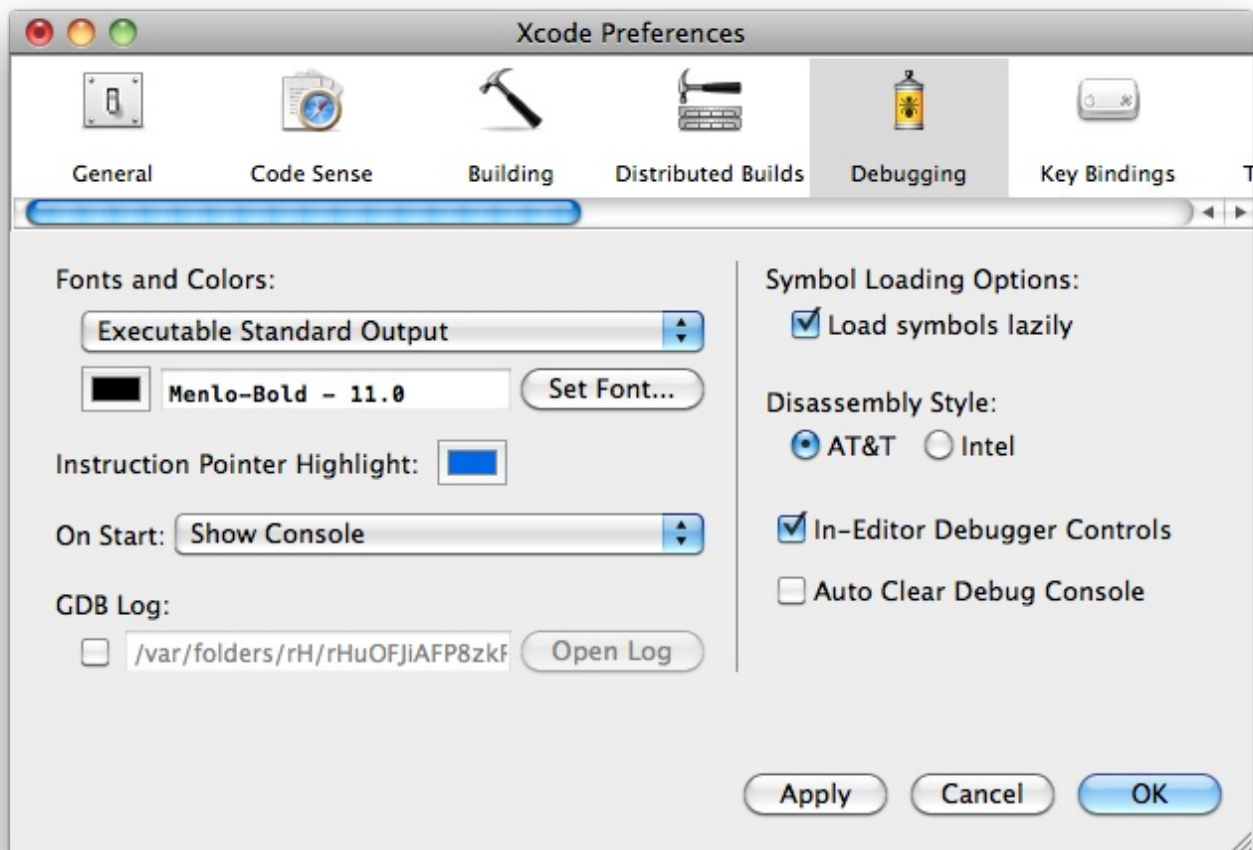


Le fichier `sdz-test` (icône noire) est l'exécutable, et le fichier `sdz-test.1` est un fichier de documentation. Le fichier `main.cpp` contient le code source du programme. Vous pouvez double-cliquer pour l'ouvrir.

Vous pouvez ajouter de nouveaux fichiers C++ au projet via le menu `File > New File`.

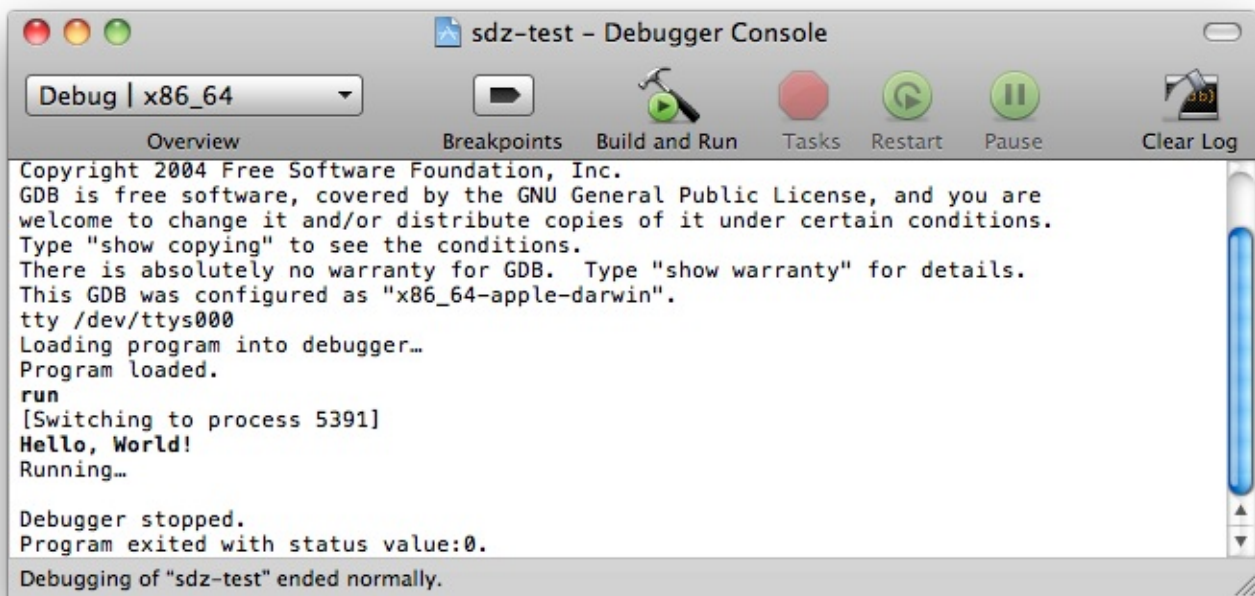
Compilation

Avant de compiler, il faut changer un réglage dans les préférences de Xcode. Pour ouvrir les préférences, il faut cliquer sur "Preferences" dans le menu "Xcode". Dans le panneau debugging, il faut sélectionner "Show console" dans le menu déroulant en face de "On start". C'est une manipulation qui est nécessaire pour voir la sortie d'un programme en console.



Cette manipulation n'a besoin d'être faite qu'une seule fois en tout.

Pour compiler, on clique sur le bouton "Build and Run" (en forme de marteau avec une petite icône verte devant) dans la fenêtre du projet. Voici donc la console qui s'affiche :



Voilà ! Vous connaissez désormais l'essentiel pour créer un nouveau projet C++ et le compiler avec Xcode. 😊
Maintenant que nous avons installé un IDE, nous avons tous les outils en main pour programmer.

...

Qu'attendez-vous ? Rendez-vous au prochain chapitre, on commence à coder ! 😊

Votre premier programme

Vous avez appris en quoi consistait la programmation et ce qu'était le C++, vous avez installé un IDE (ce logiciel qui va vous permettre de programmer) et maintenant vous vous demandez :



Bon, c'est quand qu'on commence ? 🤖

Bonne nouvelle : c'est maintenant ! 😊

Alors bien sûr, ne vous mettez pas à vous imaginer que vous allez faire des choses folles tout d'un coup. La 3D temps réel en réseau n'est pas trop au programme pour le moment ! A la place, votre objectif du chapitre sera d'arriver à afficher un message à l'écran.

Vous allez voir... c'est déjà du travail ! 😊

Le monde merveilleux de la console

Quand je vous annonce que nous allons commencer à programmer, vous vous dites sûrement "*Chouette, je vais pouvoir faire ça, ça et ça, et j'ai toujours rêvé de faire ça aussi !*". Il est de mon devoir de calmer un peu le jeu et de vous expliquer comment ça va se passer. 😊

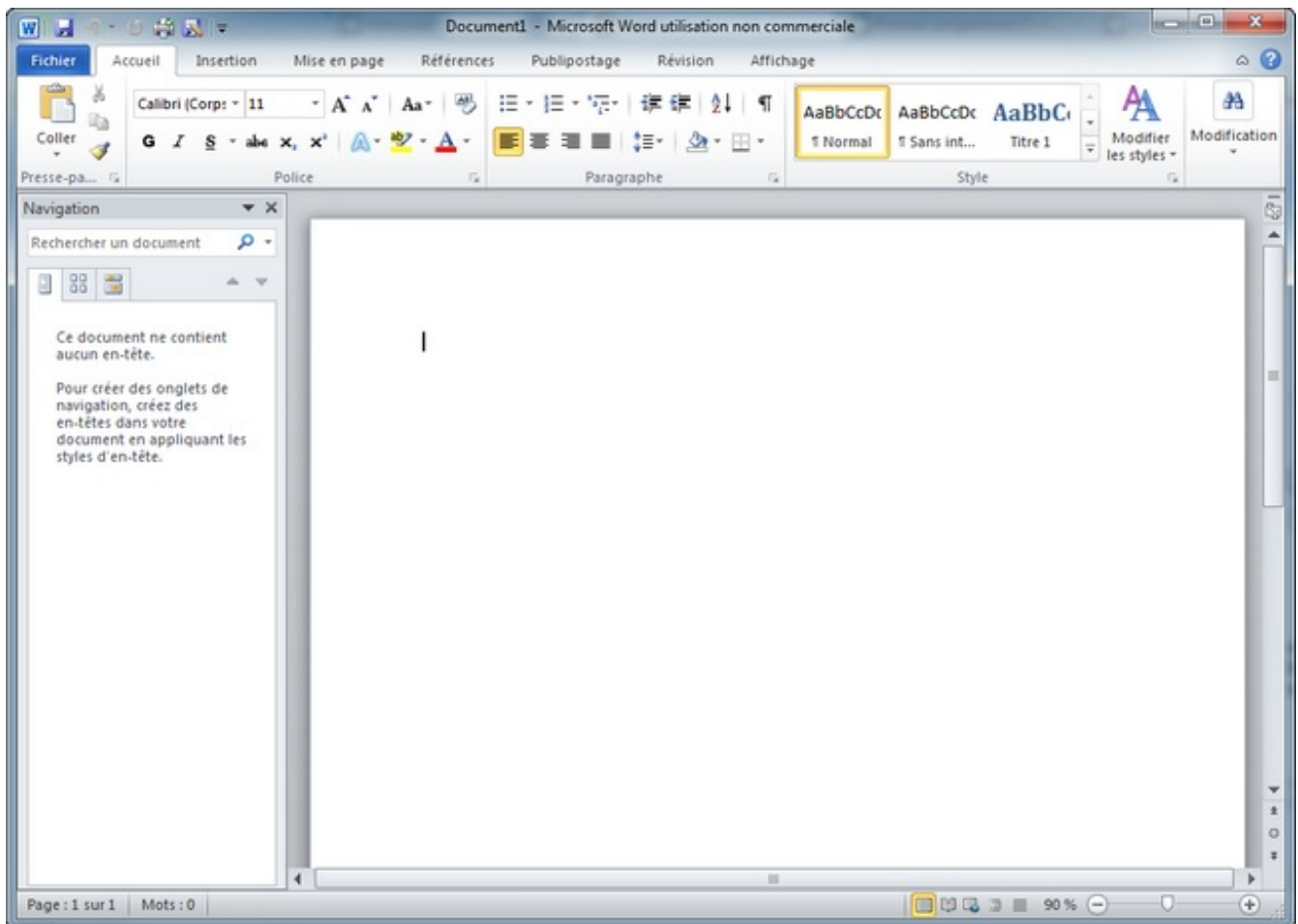
Nous allons commencer doucement. Nous n'avons de toute façon pas le choix, car les programmes complexes 3D en réseau que vous imaginez peut-être nécessitent de connaître **les bases**.

Il faut savoir qu'il existe 2 types de programmes : les programmes graphiques et les programmes console.

Les programmes graphiques

Ce sont des programmes qui affichent des fenêtres. Ce sont ceux que vous connaissez sûrement le mieux. Ils affichent des fenêtres à l'écran que l'on peut ouvrir, réduire, fermer, agrandir...

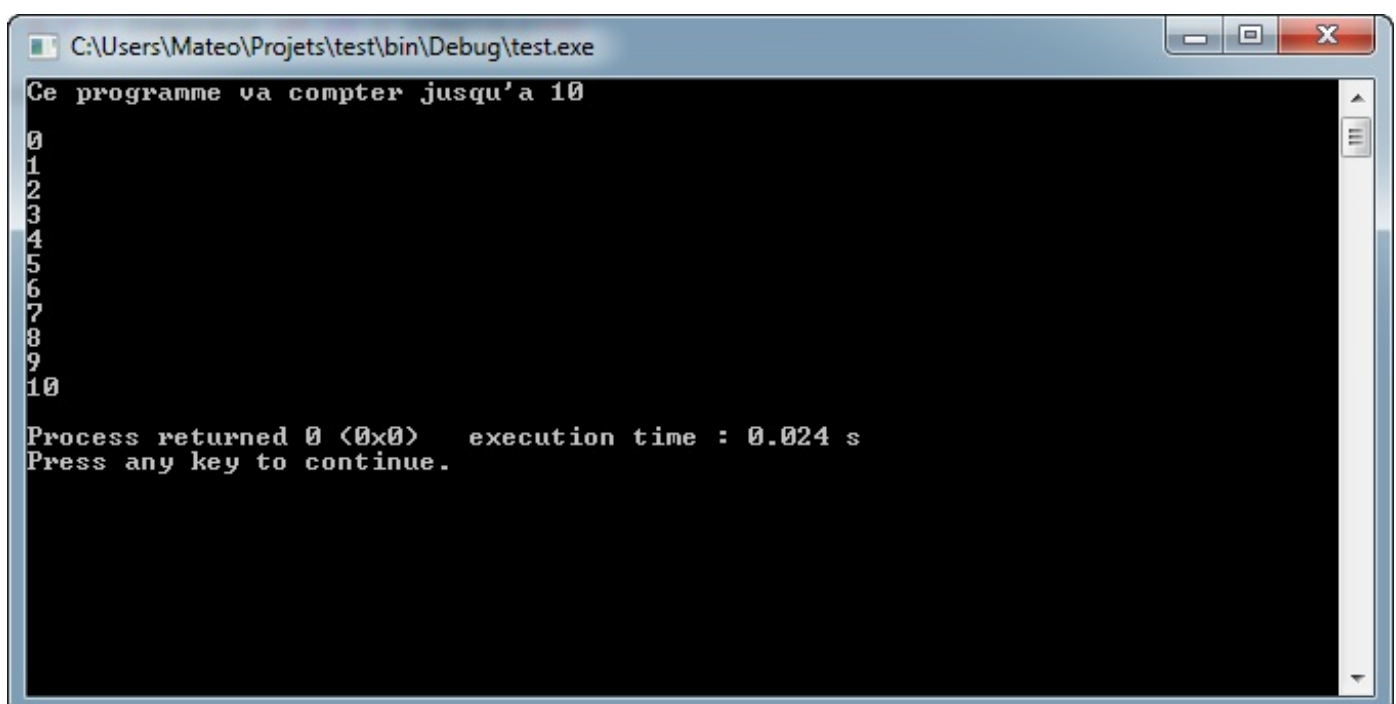
Les programmeurs parlent de GUI (Graphical User Interface - Interface Utilisateur Graphique).



Un programme GUI (graphique) : Word

Les programmes console

Les programmes en console sont plus fréquents sous Linux que sous Windows et Mac OS X. Ils sont constitués de simples textes qui s'affichent à l'écran, le plus souvent en blanc sur fond noir.



Un programme en console

Ces programmes fonctionnent au clavier. La souris n'est pas utilisée.
Ils s'exécutent généralement linéairement : les messages s'affichent au fur et à mesure de haut en bas.

Notre première cible : les programmes console

Eh oui, j'imagine que vous l'avez vue venir celle-là ! 😊

Je vous annonce que nous allons commencer par réaliser des programmes console. En effet, bien qu'un peu austères a priori, ces programmes sont beaucoup plus simples à créer que les programmes graphiques. Pour les débutants que nous sommes, il faudra donc d'abord en passer par là !

Bien entendu, je sais que vous ne voudrez pas en rester là. Rassurez-vous sur ce point : je m'en voudrais de m'arrêter aux programmes console car je sais que beaucoup d'entre vous préféreraient créer des programmes graphiques. Ca tombe bien : une partie toute entière de ce cours sera dédiée à la création de GUI avec Qt, une sorte d'extension du C++ qui permet de réaliser ce type de programmes !

Mais avant ça, il va falloir retrousser ses manches et se mettre au travail. Au boulot ! 🧐

Création et lancement d'un premier projet

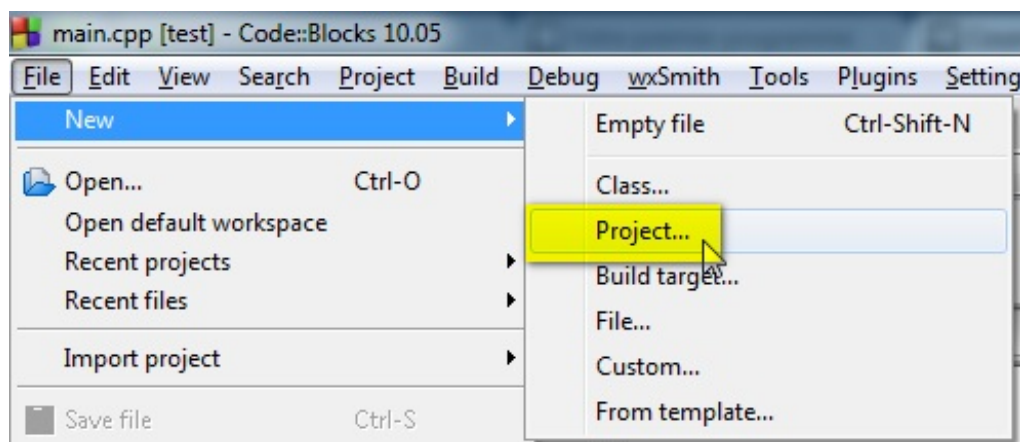
Dans le chapitre précédent, vous avez installé un IDE, ce fameux logiciel qui contient tout ce qu'il faut pour programmer. Nous avons découvert qu'il existait plusieurs IDE (Code::Blocks, Visual C++, Xcode...). Je ne vous en ai cité que quelques-uns parmi les plus connus mais il y en a bien d'autres !

Comme je vous l'avais annoncé, je travaille essentiellement sous Code::Blocks. Mes explications s'attarderont donc le plus souvent sur cet IDE, mais je reviendrai sur ses concurrents si nécessaire. Heureusement, ces logiciels se ressemblent beaucoup et emploient le même vocabulaire, donc vous ne serez pas perdus dans tous les cas. 😊

Création d'un projet

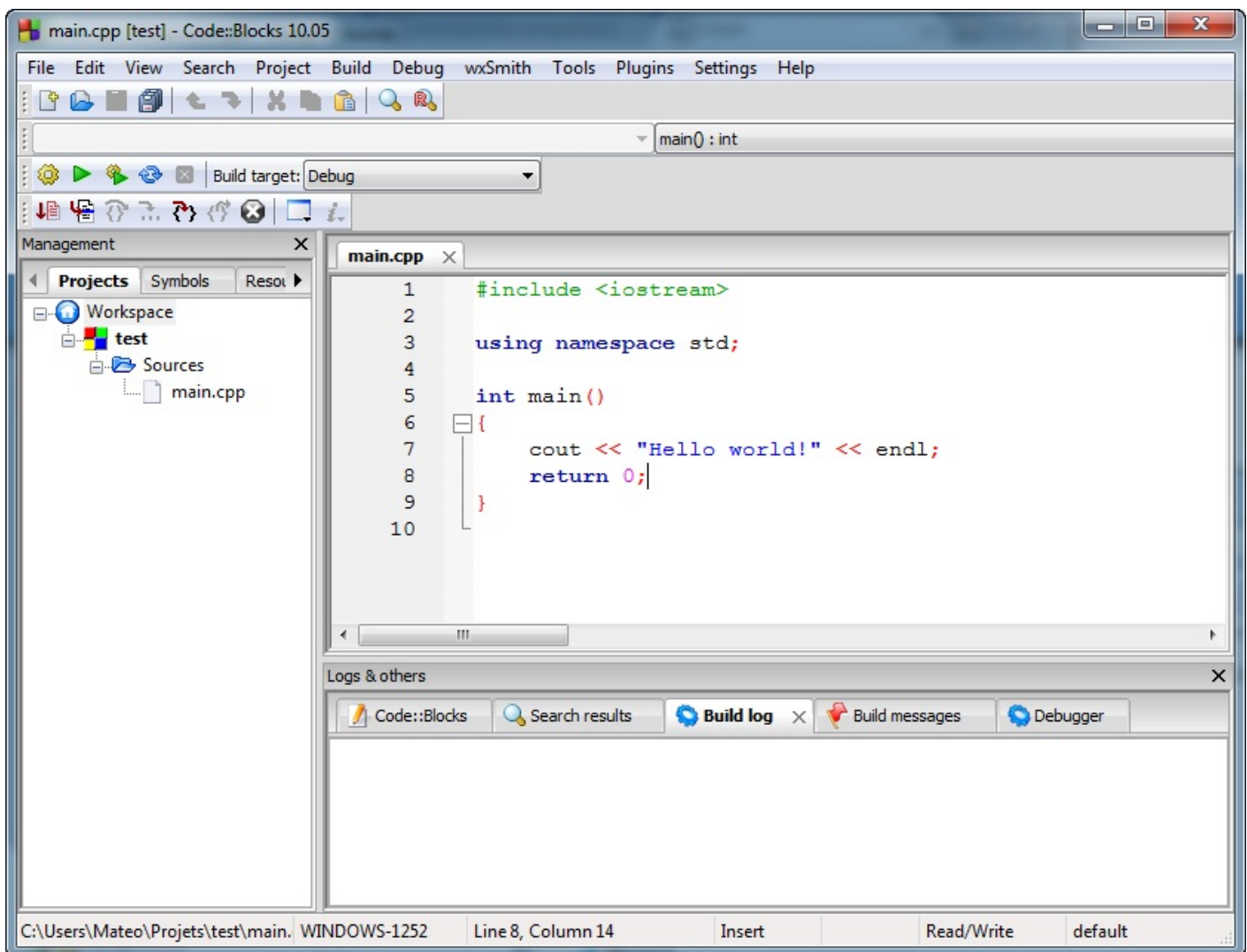
Pour commencer à programmer, la première étape consiste à demander à son IDE de créer un nouveau projet. C'est un peu comme si vous demandiez à Word de vous créer un nouveau document. 😊

Pour cela, vous allez dans le menu `File / New / Project` :



Un assistant s'ouvre, nous l'avons vu dans le chapitre précédent. Créez un nouveau programme console C++ comme nous avons appris à le faire.

A la fin des étapes de l'assistant, le projet est créé avec un premier fichier. Déroulez l'arborescence à gauche pour voir le fichier `main.cpp` apparaître et double-cliquez dessus pour l'ouvrir. Ce fichier est notre premier code source et il est déjà un peu rempli !




Code::Blocks vous a créé un premier programme très simple qui affiche le message "Hello world!" à l'écran (ça signifie quelque chose comme "Bonjour tout le monde !").

? Il y a déjà une dizaine de lignes de code source C++ et je n'y comprends rien ! 😬

Oui, ça peut paraître un peu difficile la première fois, mais nous allons voir ensemble ce que signifie ce code un peu plus loin. 😊

Lancement du programme

Pour le moment, j'aimerais que vous fassiez une chose simple : essayez de compiler et de lancer ce premier programme. Vous vous souvenez comment faire ? Il y a un bouton "Compiler et exécuter" (Build and run). Ce bouton se trouve dans la barre d'outils, dans cette section :  (c'est l'image de la roue dentée avec la flèche verte).

La compilation va alors se lancer. Vous allez voir quelques messages s'afficher en bas de l'IDE (dans la section *Build log*).

Si la compilation ne fonctionne pas et que vous avez une erreur de ce type :

Code : Console



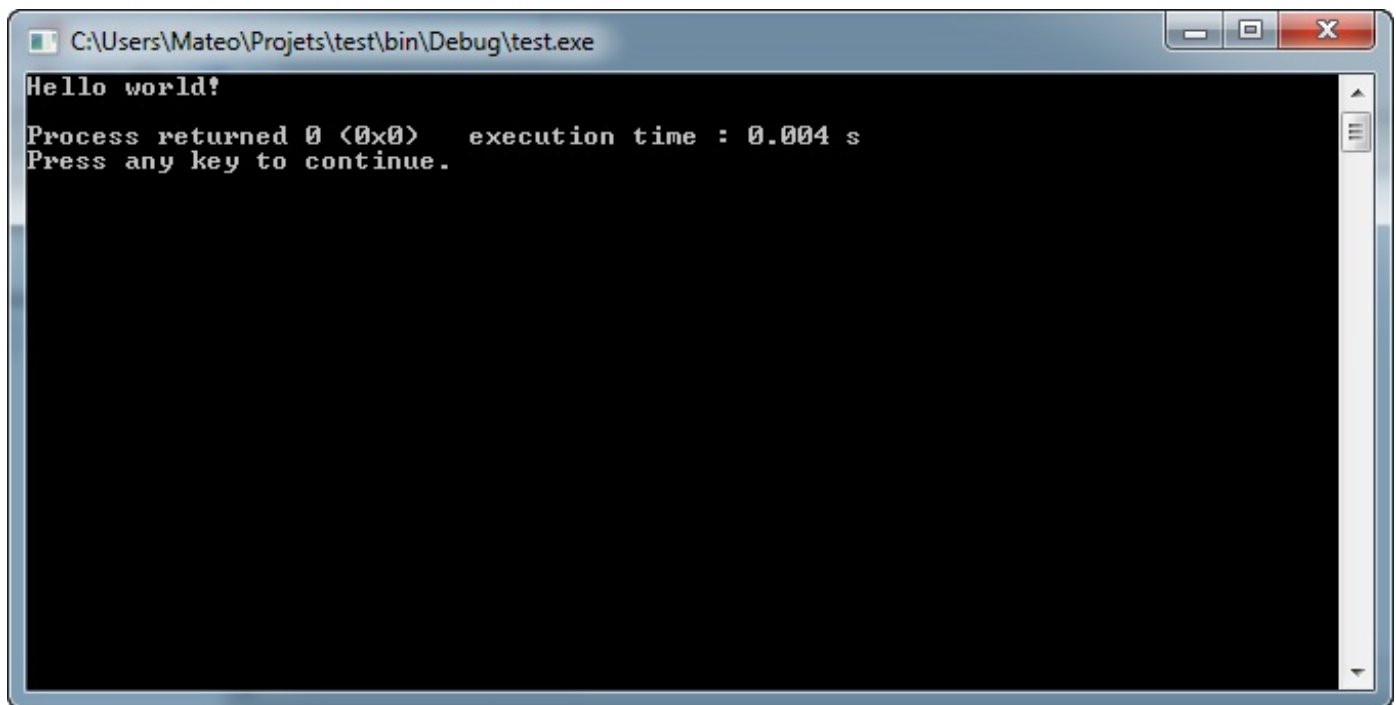
"My-program - Release" uses an invalid compiler. Skipping...
Nothing to be done.

... Cela signifie que vous avez téléchargé la version de Code::Blocks sans mingw (le compilateur). Retournez sur le site

www.siteduzero.com

de Code::Blocks pour télécharger la version avec mingw.

Si tout va bien, une console va apparaître avec notre programme :



Vous voyez que le programme affiche bel et bien "Hello world!" dans la console !
Ce n'est pas beau !? Vous venez de compiler votre tout premier programme ! 😊



Un fichier exécutable a été généré sur votre disque dur. Sous Windows, c'est un fichier .exe. Vous pouvez le retrouver dans un sous-dossier *release* (ou parfois *debug*) situé dans le dossier de votre projet.



Au fait, que signifie le message à la fin de la console :

```
Process returned 0 (0x0) execution time : 0.004 s Press any key to continue
.?
```

Ah bonne question ! 😊

Ce message n'a pas été écrit par votre programme mais par votre IDE. En l'occurrence, c'est Code::Blocks qui affiche un message pour signaler que le programme s'est bien déroulé et le temps que son exécution a duré.

Le but de Code::Blocks est ici surtout de "maintenir" la console ouverte. En effet, sous Windows en particulier, dès qu'un programme console est terminé la fenêtre de la console se ferme. Or, le programme s'étant exécuté en 0.004s ici, vous n'auriez pas eu le temps de voir le message s'afficher à l'écran !

Code::Blocks vous invite donc à "appuyer sur n'importe quelle touche pour continuer", ce qui aura pour effet de fermer la console.



Lorsque vous compilez et exécutez un programme "console" comme celui-ci avec Visual C++, la console a tendance à s'ouvrir et se refermer instantanément. Visual C++ ne maintient pas la console ouverte comme Code::Blocks. La solution consiste à ajouter la ligne `system("PAUSE")` ; avant le `return 0` ; de votre programme si vous utilisez Visual C++.

Explications du premier code source

Lorsque Code::Blocks a créé un nouveau projet, il a créé un fichier `main.cpp` contenant ce code :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```



Tous les IDE proposent en général de démarrer avec un code similaire. Cela permet de commencer à programmer plus vite. Vous retrouverez les 3 premières lignes (include, using namespace et int main) dans quasiment tous vos programmes C++. Vous pouvez considérer que tous vos programmes commenceront avec ces lignes.

Sans trop rentrer dans les détails (car cela pourrait devenir compliqué pour un début !), je vais vous présenter à quoi servent chacune de ces lignes. Vous les retrouverez dans la plupart de vos programmes.

include

La toute première ligne est :

Code : C++

```
#include <iostream>
```

C'est ce qu'on appelle une *directive de préprocesseur*. Son rôle est de "charger" des fonctionnalités du C++ pour que nous puissions effectuer certaines actions.

En effet, *le C++ est un langage très modulaire*. De base, il ne sait pas faire grand-chose (pas même afficher un message à l'écran !). On doit charger des extensions que l'on appelle **bibliothèques** et qui nous donnent de nouvelles possibilités.

Ici, on charge le fichier `iostream`, ce qui nous permet d'utiliser une bibliothèque... d'affichage de messages à l'écran dans une console ! Quelque chose de vraiment très basique, comme vous le voyez, mais qui nécessite quand même l'utilisation d'une bibliothèque.



Appeler `iostream` nous permet en fait de faire un peu plus que d'afficher des messages à l'écran : on pourra aussi récupérer ce que saisit l'utilisateur au clavier comme nous le verrons plus tard. `iostream` signifie "Input Output Stream", ce qui signifie "Flux d'entrée-sortie". Dans un ordinateur, l'entrée correspond en général au clavier (ou la souris), et la sortie à l'écran. Inclure `iostream` nous permet donc en quelque sorte d'obtenir tout ce qu'il faut pour échanger des informations avec l'utilisateur.

Plus tard, nous découvrirons de nouvelles bibliothèques et il faudra effectuer des inclusions en haut des codes source comme ici. Par exemple, lorsque nous étudierons Qt qui permet de réaliser des programmes graphiques (GUI), on insérera une ligne comme celle-ci :

Code : C++

```
#include <Qt>
```

Notez qu'on peut charger autant de bibliothèques que l'on veut à la fois.

using namespace

La ligne :

Code : C++

```
using namespace std;
```

... permet en quelque sorte d'indiquer dans quel lot de fonctionnalités notre fichier source va aller piocher.

Si vous chargez plusieurs bibliothèques, chacune va proposer de nombreuses fonctionnalités. Parfois, certaines fonctionnalités ont le même nom. Imaginez une commande "AfficherMessage" qui s'appelle ainsi pour iostream mais aussi pour Qt ! Si vous chargez les deux bibliothèques en même temps et que vous appelez "AfficherMessage", l'ordinateur ne saura pas s'il doit afficher un message en console avec iostream ou dans une fenêtre avec Qt !

Pour éviter ce genre de problèmes, on a créé des namespaces (espaces de noms) qui sont des sortes de dossiers à noms. La ligne `using namespace std;` indique que vous allez utiliser l'espace de noms std dans la suite de votre fichier de code. Cet espace de noms est un des plus connus car il correspond à la bibliothèque standard (std), une bibliothèque livrée par défaut avec le langage C++ et dont iostream fait partie.

int main()

C'est ici que commence vraiment le coeur du programme. Les programmes, vous le verrez, sont essentiellement constitués de fonctions. Chaque fonction a un rôle et peut en appeler d'autres pour effectuer certaines actions.

Tous les programmes possèdent une fonction qui s'appelle "main" (prononcez en anglais "mèine"), ce qui signifie "principale". C'est donc la fonction principale. 🤔

Une fonction a cette forme :

Code : C++

```
int main()  
{  
  
}
```

Les accolades déterminent le début et la fin de la fonction. Comme vous le voyez dans le code source qui nous a été généré par Code::Blocks, il n'y a rien après la fonction main. C'est normal : à la fin de la fonction main le programme s'arrête ! Tout programme commence au début de la fonction main et termine à la fin de celle-ci.



Cela veut dire qu'on va écrire tout notre programme dans la fonction main ? 🤔

Non ! Bien que ce soit possible, ce serait très délicat à gérer surtout pour de gros programmes. A la place, la fonction main appelle d'autres fonctions qui à leur tour appellent d'autres fonctions. Bref, elle délègue le travail.

Dans un premier temps cependant, nous allons surtout travailler dans la fonction main car nos programmes seront assez simples pour commencer.

cout

Voici enfin la première ligne qui fait quelque chose de concret ! C'est la première ligne du main, donc la première action qui sera exécutée par l'ordinateur (ce que nous avons vu précédemment ne sont en fait que des préparatifs pour le programme).

Code : C++

```
cout << "Hello world!" << endl;
```

Le rôle de `cout` (prononcez "ci aoute") est d'afficher un message à l'écran. C'est ce qu'on appelle une **instruction**. Tous nos programmes seront constitués d'instructions comme celle-ci qui donnent des ordres à l'ordinateur.

Notez que `cout` est fourni par `iostream`. Si vous n'incluez pas `iostream` au début de votre programme, le compilateur dira qu'il ne connaît pas `cout` et vous ne pourrez pas générer votre programme !



Notez bien : chaque instruction se termine par un point-virgule ! C'est d'ailleurs ce qui vous permet de différencier les instructions du reste. 😊

Si vous oubliez le point-virgule, la compilation ne fonctionnera pas et votre programme ne pourra pas être créé !

Il y a 3 éléments sur cette ligne :

- `cout` : commande l'affichage d'un message à l'écran
- `"Hello world!"` : indique le message à afficher
- `endl` : crée un retour à la ligne dans la console

Il est possible de combiner plusieurs messages en une instruction. Par exemple :

Code : C++

```
cout << "Bonjour tout le monde !" << endl << "Comment allez-vous ?" << endl;
```

... affichera ces deux phrases sur 2 lignes différentes. Essayez ce code, vous verrez !



Sous Windows, les caractères accentués s'affichent mal (essayez d'afficher "Bonjour Gérard" pour voir !). C'est un problème de la console de Windows (problème qu'on peut retrouver plus rarement sous Mac OS X et Linux). Il existe des moyens de le régler mais aucun n'est vraiment satisfaisant. A la place, je vous recommande plutôt d'éviter les accents dans les programmes console sous Windows.

Rassurez-vous : les GUI que nous créerons plus tard avec Qt n'auront pas ce problème !

return

La dernière ligne est :

Code : C++

```
return 0;
```

Ce type d'instruction clôt généralement les fonctions. En fait, la plupart des fonctions renvoient une valeur (un nombre par exemple). Ici, la fonction `main` renvoie 0 pour indiquer que tout s'est bien passé (toute valeur différente de 0 aurait indiqué un problème).

Vous n'avez pas besoin de modifier cette ligne, laissez-la telle quelle. Nous aurons l'occasion d'utiliser `return` d'autres fois pour d'autres fonctions, nous en reparlerons !

Commentez vos programmes !

En plus du code qui donne des instructions à l'ordinateur, vous pouvez écrire des commentaires pour expliquer le

fonctionnement de votre programme.

Les commentaires n'ont aucun impact sur le fonctionnement de votre logiciel : en fait, le compilateur ne les lit même pas et ils n'apparaissent pas dans le programme généré. Pourtant, ces commentaires sont indispensables pour les développeurs : ils leur permettent d'expliquer ce qu'ils font dans leur code !

Dès que vos programmes vont devenir un petit peu complexes (et croyez-moi, ça ne tardera pas 😊), vous risquez d'avoir du mal à vous souvenir de leur fonctionnement quelque temps après avoir écrit le code source. De plus, si vous envoyez votre code à un ami, il aura des difficultés à comprendre ce que vous avez essayé de faire juste en lisant le code source. C'est là que les commentaires entrent en jeu !

Les différents types de commentaires

Il y a 2 façons d'écrire des commentaires, selon leur longueur. Je vais vous les présenter toutes les deux.

Les commentaires courts

Pour écrire un commentaire court, sur une seule ligne, il suffit de commencer par `//` puis d'écrire votre commentaire. Cela donne :

Code : C++

```
// Ceci est un commentaire
```

Mieux, vous pouvez aussi ajouter le commentaire à la fin d'une ligne pour expliquer ce qu'elle fait :

Code : C++

```
cout << "Hello world!" << endl; // Affiche un message à l'écran
```

Les commentaires longs

Si votre commentaire tient sur plusieurs lignes, ouvrez la zone de commentaire avec `/*` et fermez-la avec `*/` :

Code : C++

```
/* Le code qui suit est un peu complexe  
alors je prends mon temps pour l'expliquer  
parce que je sais que sinon dans quelques semaines  
j'aurai tout oublié et je serai perdu pour le modifier */
```

En général, on n'écrit pas un roman dans les commentaires non plus... sauf si la situation le justifie vraiment.

Commentons notre code source !

Reprenons le code source que nous avons étudié dans ce chapitre et complétons-le de quelques commentaires pour nous souvenir de ce qu'il fait.

Code : C++

```
#include <iostream> // Inclut la bibliothèque iostream (affichage  
de texte)
```

```
using namespace std; // Indique quel espace de noms on va utiliser

/*
Fonction principale "main"
Tous les programmes commencent par la fonction main
*/
int main()
{
    cout << "Hello world!" << endl; // Affiche un message
    return 0; // Termine la fonction main et donc le programme
}
```

Si vous lancez ce programme, vous ne verrez aucune nouveauté. Les commentaires sont, comme je vous le disais, purement ignorés par le compilateur.



J'ai volontairement commenté chaque ligne de code ici, mais dans la pratique il ne faut pas commenter à tout-va non plus. Si une ligne de code fait quelque chose de vraiment évident, inutile de la commenter. En fait, les commentaires sont plus utiles pour expliquer le fonctionnement d'une série d'instructions, plutôt que chaque instruction une à une.

Vous avez mis en place votre tout premier programme : bravo !

Pour le moment, vous savez seulement afficher un message à l'écran, mais vous allez pouvoir aller de plus en plus loin au fur et à mesure des chapitres qui vont venir. Dans le prochain chapitre, nous allons commencer à manipuler la mémoire ! 😊

Utiliser la mémoire

Jusqu'à présent, vous avez découvert comment créer vos premiers programmes en mode console. Vous avez aussi appris à les compiler, ce qui n'a pas été une mince affaire. Pour l'instant les programmes que vous avez réalisés sont très simples. Ils affichent des messages à l'écran et c'est un peu tout. 😞

Je suis d'accord avec vous, ce n'est pas assez pour faire quelque chose d'intéressant. Cela est principalement dû au fait que vos programmes ne savent pas interagir avec leurs utilisateurs. C'est ce que nous allons apprendre à faire dans le chapitre suivant, puisque je vais vous montrer comment demander des informations à l'utilisateur. Nous pourrons ainsi écrire notre premier programme interactif.

Mais avant ça, il va nous falloir travailler dur, puisque je vais vous présenter une notion fondamentale en informatique. Nous allons parler des **variables**.

L'idée de base qui se cache derrière la notion de variable c'est de mettre quelque chose dans la mémoire de l'ordinateur afin de le ré-utiliser plus tard. J'imagine que vous avez tous déjà eu une calculatrice entre les mains. Sur ces outils, il y a généralement des touches M+, M-, MC, etc. qui permettent de stocker un résultat intermédiaire d'un calcul dans la mémoire de la calculatrice et de reprendre ce nombre plus tard. Nous allons apprendre à faire la même chose avec votre ordinateur, qui n'est après tout qu'une grosse machine à calculer. 😞



Une fois que nous aurons appris à déclarer des variables, nous pourrons les utiliser pour interagir avec les utilisateurs de nos programmes et leur demander par exemple, leur nom ou leur âge.

Qu'est-ce qu'une variable ?

Je vous ai donné l'exemple de la mémoire de la calculatrice avant parce que dans le monde de l'informatique le principe de base est le même. Il y a quelque part, dans votre ordinateur, des composants électroniques qui sont capables de contenir une valeur et de la conserver pendant un certain temps. La manière dont tout cela fonctionne exactement est très complexe. 😞

Je vous rassure tout de suite, on n'a absolument pas besoin de comprendre comment ça marche pour pouvoir, nous aussi, mettre des valeurs dans la mémoire du PC. Toute la partie compliquée sera gérée par le compilateur et le système d'exploitation. Elle est pas belle la vie ?

La seule et unique chose que vous avez besoin de savoir, c'est qu'une **variable** est une partie de la mémoire que l'ordinateur nous prête pour y mettre des valeurs. Imaginez que l'ordinateur possède dans ses entrailles une grande armoire. Cette armoire possède des milliers (des milliards) de petits tiroirs, ce sont des endroits que nous allons pouvoir utiliser pour mettre nos variables.

Dans le cas d'une calculatrice toute simple, on ne peut généralement stocker qu'un seul nombre à la fois. Vous vous doutez bien que dans le cas d'un programme, il va falloir conserver plus d'une chose simultanément. Il faut donc un moyen de différencier les variables pour pouvoir par la suite y accéder. Chaque variable possède donc un **nom**. C'est en quelque sorte l'étiquette qui est collée sur le tiroir.

L'autre chose qui distingue la calculatrice de l'ordinateur, c'est que nous aimerions pouvoir stocker des tas de choses différentes, des nombres, des lettres, des phrases, des images, etc. C'est ce qu'on appelle le **type** d'une variable. Vous pouvez vous imaginer cela comme étant la forme du tiroir. On utilise en effet pas les mêmes tiroirs pour stocker des bouteilles ou des livres.



Les noms de variables

Commençons par la question du nom des variables. En C++, il y a quelques règles qui régissent les différents noms autorisés ou interdits.

- Les noms de variables sont constitués de lettres, de chiffres et du tiret-bas _ uniquement.
- Le premier caractère doit être une lettre (majuscule ou minuscule).
- On ne peut pas utiliser d'accents.
- On ne peut pas utiliser d'espaces dans le nom.

Le mieux est encore de vous donner quelques exemples. Les noms `ageZero`, `nom_du_zero` ou encore `NOMBRE_ZEROS` sont tous des noms valides. `AgeZéro`, `_nomzero` ne le sont par contre pas.

A cela s'ajoute une règle supplémentaire qui est valable pour tout ce que l'on écrit en C++ et pas seulement pour les variables. Le langage fait la différence entre les majuscules et les minuscules. En termes techniques, on dit que C++ est *sensible à la casse*. Donc, `nomZero`, `nomzero`, `NOMZERO` et `NomZeRo` sont tous des noms de variables différents.



Pour des questions de lisibilité, il est important d'utiliser des noms de variables qui décrivent bien ce qu'elles contiennent. On préférera donc choisir `ageUtilisateur` comme nom plutôt que `maVar` ou `variable1`. Pour le compilateur, cela ne joue aucun rôle. Mais pour vous et pour les gens qui travailleront avec vous sur le même programme, c'est très important.

Personnellement, j'utilise une "convention" partagée par beaucoup de programmeurs. Dans tous les gros projets regroupant des milliers de programmeurs on trouve des règles très strictes et parfois difficiles à suivre. Celles que je vous propose ici permettent de garder une bonne lisibilité et surtout vous permettront de bien comprendre tous les exemples dans la suite de ce cours.

- Les noms de variables commencent par une minuscule.
- Si le nom se décompose en plusieurs mots, ceux-ci sont collés les uns aux autres.
- Chaque nouveau mot (excepté le premier) commence par une majuscule.

Voyons ça avec des exemples. Prenons le cas d'une variable censée contenir l'âge de l'utilisateur du programme.

- `AgeUtilisateur`: Non, car la première lettre est une majuscule.
- `age_utilisateur`: Non, car les mots ne sont pas collés
- `ageutilisateur`: Non, car le deuxième mot ne commence pas par une majuscule.
- `maVar`: Non, car le nom ne décrit pas ce que contient la variable.
- `ageUtilisateur`: Ok.

Je vous conseille fortement d'utiliser la même convention. Rendre son code lisible et facilement compréhensible par d'autres programmeurs est très important.

Les types de variables

Reprenons. Nous avons appris qu'une variable a un nom et un type. Nous savons comment nommer nos variables, voyons maintenant leurs différents types. L'ordinateur aime savoir ce qu'il a dans sa mémoire, il faut donc indiquer quel type d'élément va contenir la variable que nous aimerions utiliser. Est-ce un nombre, un mot, une lettre ? Il faut le spécifier.

Voici donc la liste des types de variables que l'on peut utiliser en C++.

Nom du type	Ce qu'il peut contenir
<code>bool</code>	Peut contenir deux valeurs "vrai" (<code>true</code>) ou "faux" (<code>false</code>).
<code>char</code>	Une lettre.
<code>int</code>	Un nombre entier.
<code>unsigned int</code>	Un nombre entier positif ou nul.
<code>double</code>	Un nombre à virgule.
<code>string</code>	Une chaîne de caractères. C'est-à-dire une suite de lettres, un mot, une phrase.

Si vous tapez un de ces noms de type dans votre IDE, vous devriez voir le mot se colorer. L'IDE l'a reconnu, c'est bien la preuve que je ne vous raconte pas des salades. Le cas de `string` est différent; nous verrons plus loin pourquoi. Je peux vous assurer qu'on va beaucoup en parler. 🤪



Ces types ont des limites de validité, des bornes. C'est-à-dire qu'il y a des nombres qui sont trop grand pour un `int` par exemple. Ces bornes dépendent de votre ordinateur, de votre système d'exploitation et de votre compilateur. Sachez simplement que ces limites sont bien assez grandes pour la plupart des utilisations courantes. Cela ne devrait donc pas vous poser de problèmes à moins que vous vouliez créer des programmes pour téléphones portables ou pour des micro-contrôleurs qui ont parfois des bornes plus basses que les PCs. 😊 Il existe également d'autres types avec d'autres limites, mais ils sont utilisés plus rarement.

Quand on a besoin d'une variable, il faut donc se poser la question du genre de choses qu'elle va contenir. Si vous avez besoin d'une variable pour stocker le nombre de personnes qui utilisent votre programme, alors utilisez un `int` ou `unsigned int`, pour stocker le poids d'un gigot, on utilisera un `double` et pour conserver en mémoire le nom de votre meilleur ami, on choisira une chaîne de caractères `string`.



Mais, à quoi sert le type `bool` ? J'en ai jamais entendu parler.

C'est ce qu'on appelle un booléen. C'est-à-dire une variable qui ne peut prendre que deux valeurs, vrai (`true` en anglais) ou faux (`false` en anglais). On les utilise par exemple pour stocker des informations comme, la lumière est-elle allumée ? L'utilisateur a-t-il le droit d'utiliser cette fonctionnalité ? Le mot de passe est-il correct ?

Si vous avez besoin de conserver le résultat d'une question de ce genre, alors pensez à ce type de variable.

Déclarer une variable

Assez parlé, il est temps d'entrer dans le vif du sujet et de demander à l'ordinateur de nous prêter un de ses tiroirs. En terme technique, on parle de **déclaration de variable**.

Il nous faut indiquer à l'ordinateur, le type de la variable que l'on veut, son nom et enfin sa valeur. Pour se faire, c'est très simple. On indique les choses exactement dans cet ordre.

TYPE NOM (VALEUR);

On peut aussi utiliser la même syntaxe que dans le langage C:

TYPE NOM = VALEUR ;

Les deux versions sont *strictement équivalentes*. Je vous conseille cependant d'utiliser la première pour des raisons qui deviendront claires plus tard. La deuxième version ne sera pas utilisée dans la suite du cours, je vous l'ai mise pour que vous puissiez comprendre les nombreux exemples que l'on peut trouver sur le web et qui utilisent cette version de la déclaration d'une variable.



N'oubliez pas le point-virgule (;) à la fin de la ligne ! C'est le genre de choses que l'on oublie très facilement et le compilateur n'aime pas ça du tout. 🤖

Reprenons le morceau de code minimal et ajoutons-y une variable pour stocker l'âge de l'utilisateur.

Code : C++ - Déclaration d'une variable

```
#include <iostream>
using namespace std;

int main()
{
    int ageUtilisateur(16);
    return 0;
}
```

Que se passe-t-il à la ligne 6 de ce programme ?

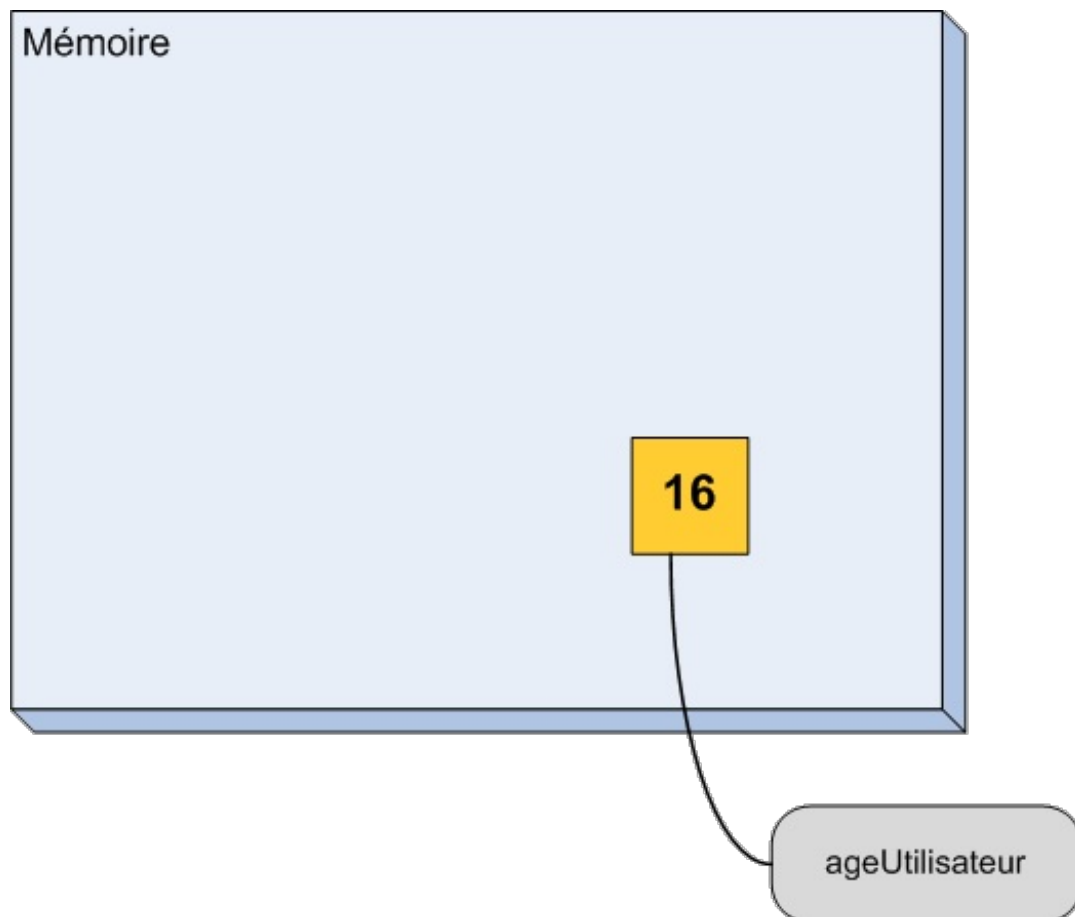
L'ordinateur voit que l'on aimerait lui emprunter un tiroir dans sa mémoire avec les propriétés suivantes :

- Il peut contenir des nombres entiers.
- Il a une étiquette indiquant qu'il s'appelle `ageUtilisateur`.
- Il contient la valeur **16**.

A partir de cette ligne, vous êtes donc l'heureux possesseur d'un tiroir dans la mémoire de l'ordinateur. 😊



Comme nous allons avoir besoin de beaucoup de tiroirs dans la suite du cours, je vous propose d'utiliser des schémas un peu plus simples. On va beaucoup les utiliser par la suite, il est donc bien de s'y habituer tôt.



Je vais vous décrire un peu ce qu'on voit sur le schéma. Le gros rectangle bleu représente la mémoire de l'ordinateur. Pour l'instant, elle est presque vide. Le carré jaune est la zone de mémoire que l'ordinateur vous a prêtée. C'est l'équivalent de notre tiroir. Il contient, comme avant, le nombre **16** et on peut lire le nom `ageUtilisateur` sur l'étiquette qui y est accrochée. Je ne suis pas bon en dessin, donc il faut un peu imaginer hein. 😊

Ne nous arrêtons pas en si bon chemin. Déclarons d'autres variables.

Code : C++ - Un amour de déclaration

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int ageUtilisateur(16);
    int nombreAmis(432);           //Le nombre d'amis de l'utilisateur

    double pi(3.14159);

    bool estMonAmi(true);         //Cet utilisateur est-il mon ami ?

    char lettre('a');

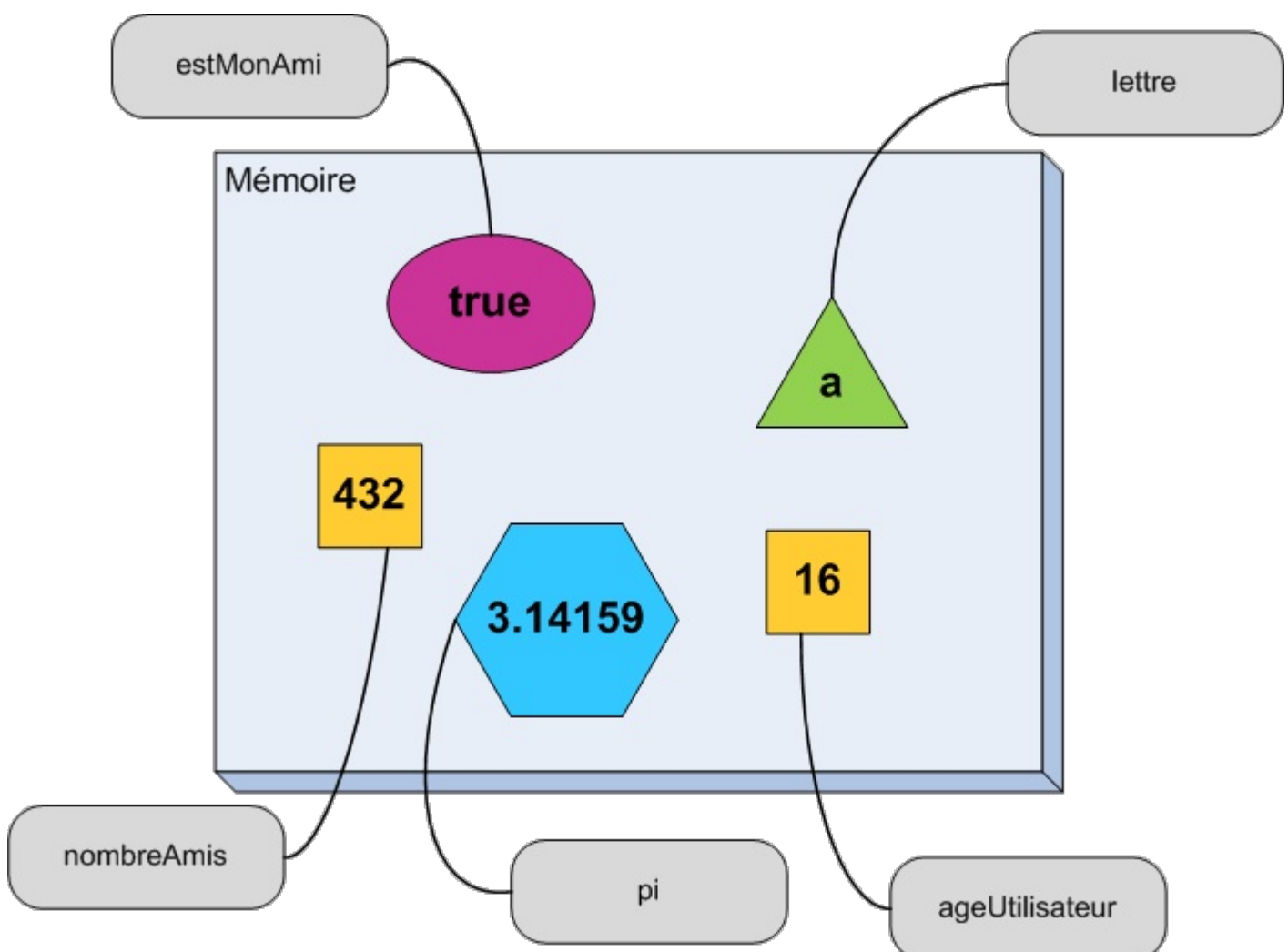
    return 0;
}
```

Il y a deux choses importantes à remarquer ici, la première est que les variables de type `bool` ne peuvent avoir pour valeur que `true` ou `false`. C'est donc une de ces deux valeurs qu'il faut mettre entre les parenthèses. L'autre chose dont il faut se souvenir, c'est que pour le type `char`, il faut mettre la lettre que l'on veut entre apostrophes. Il faut écrire `char lettre('a');` et pas `char lettre(a);`. C'est une erreur que tout le monde fait, moi le premier. 😊



Il est toujours bien de mettre un commentaire pour expliquer à quoi va servir la variable.

Je peux donc compléter mon schéma en lui ajoutant nos nouvelles variables.



Vous pouvez évidemment compiler et tester le programme ci-dessus. Vous constaterez qu'il ne fait strictement rien. J'espère que vous n'êtes pas trop déçu. Il se passe en réalité énormément de choses mais comme je vous l'ai dit au début, ces opérations sont

cachées et ne nous intéressent pas vraiment. En voici quand même un résumé chronologique.

1. Votre programme demande au système d'exploitation de lui fournir un peu de mémoire.
2. L'OS a regardé s'il en avait encore à disposition et a indiqué au programme quel tiroir utiliser.
3. Le programme a écrit la valeur **16** dans la case mémoire.
4. Il a ensuite recommencé pour les quatre autres variables.
5. En arrivant à la dernière ligne, le programme a vidé ses tiroirs et les a rendus à l'ordinateur.

Et tout ça sans que rien ne se passe du tout à l'écran ! C'est normal, on n'a nulle part indiqué qu'on voulait afficher quelque chose.

Le cas des strings

Les chaînes de caractères sont un petit peu plus complexes à déclarer, mais rien d'insurmontable, je vous rassure. La première chose à faire est d'ajouter une petite ligne au début de votre programme. Il faut, en effet, indiquer au compilateur que nous souhaitons utiliser des strings. Sans ça, il n'inclurait pas les outils nécessaires à leur gestion. La ligne à ajouter est `#include <string>`.

Voici ce que ça donne.

Code : C++ - Votre premier string

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string nomUtilisateur("Albert Einstein");
    return 0;
}
```

L'autre différence se situe au niveau de la déclaration elle-même. Comme vous l'avez certainement constaté, j'ai utilisé des guillemets autour de la valeur. Un peu comme pour les lettres, mais cette fois ce sont des guillemets doubles (") et pas juste des apostrophes ('). D'ailleurs votre IDE devrait colorier les mots "Albert Einstein" d'une couleur différente du 'a' de l'exemple précédent, même si ce n'est pas le cas sur le site du zéro. Confondre ' et " est une erreur à nouveau très courante qui fera hurler de douleur votre compilateur. Mais ne vous en faites pas pour lui, il en a vu d'autres. 😊

Une astuce pour gagner de la place

Avant de passer à la suite, il faut que je vous présente une petite astuce utilisée par certains programmeurs.

Si vous avez plusieurs variables du **même type** à déclarer, vous pouvez le faire sur une seule ligne en les séparant par une virgule (,). Voici comment:

Code : C++ - Déclarer plusieurs variables sur une seule ligne

```
int a(2), b(4), c(-1); //On déclare trois cases mémoires nommées a, b
et c et qui contiennent les valeurs 2, 4 et -1 respectivement.

string prenom("Albert"), nom("Einstein"); //On déclare deux cases
pouvant contenir des chaînes de caractères
```

Ça peut être pratique quand on a besoin de beaucoup de variables d'un coup. On économise l'écriture du type à chaque fois mais, je vous déconseille quand même de trop abuser de cette astuce. Le programme devient moins lisible et moins compréhensible.

Déclarer sans initialiser

Maintenant que nous avons vu le principe général, il est temps de plonger un petit peu plus dans les détails.

Lors de la déclaration d'une variable, votre programme effectue en réalité deux opérations successives.

1. Il demande à l'ordinateur de lui fournir une zone de stockage dans la mémoire.
2. Il remplit cette case avec la valeur fournie. On parle alors d'**initialisation** de la variable.

Ces deux étapes s'effectuent automatiquement et sans que l'on ait besoin de rien faire. Voilà pour la partie vocabulaire de ce chapitre. 😊

Il arrive parfois que l'on ne sache pas quelle valeur donner à une variable lors de sa déclaration. Il est alors possible d'effectuer uniquement l'allocation sans l'initialisation.

Il suffit d'indiquer le **type** et le **nom** de la variable sans spécifier de valeur.

TYPE NOM ;

Et sous forme de code C++ complet, voilà ce que ça donne :

Code : C++ - Déclaration sans initialisation

```
#include <iostream>
#include <string>
using namespace std;

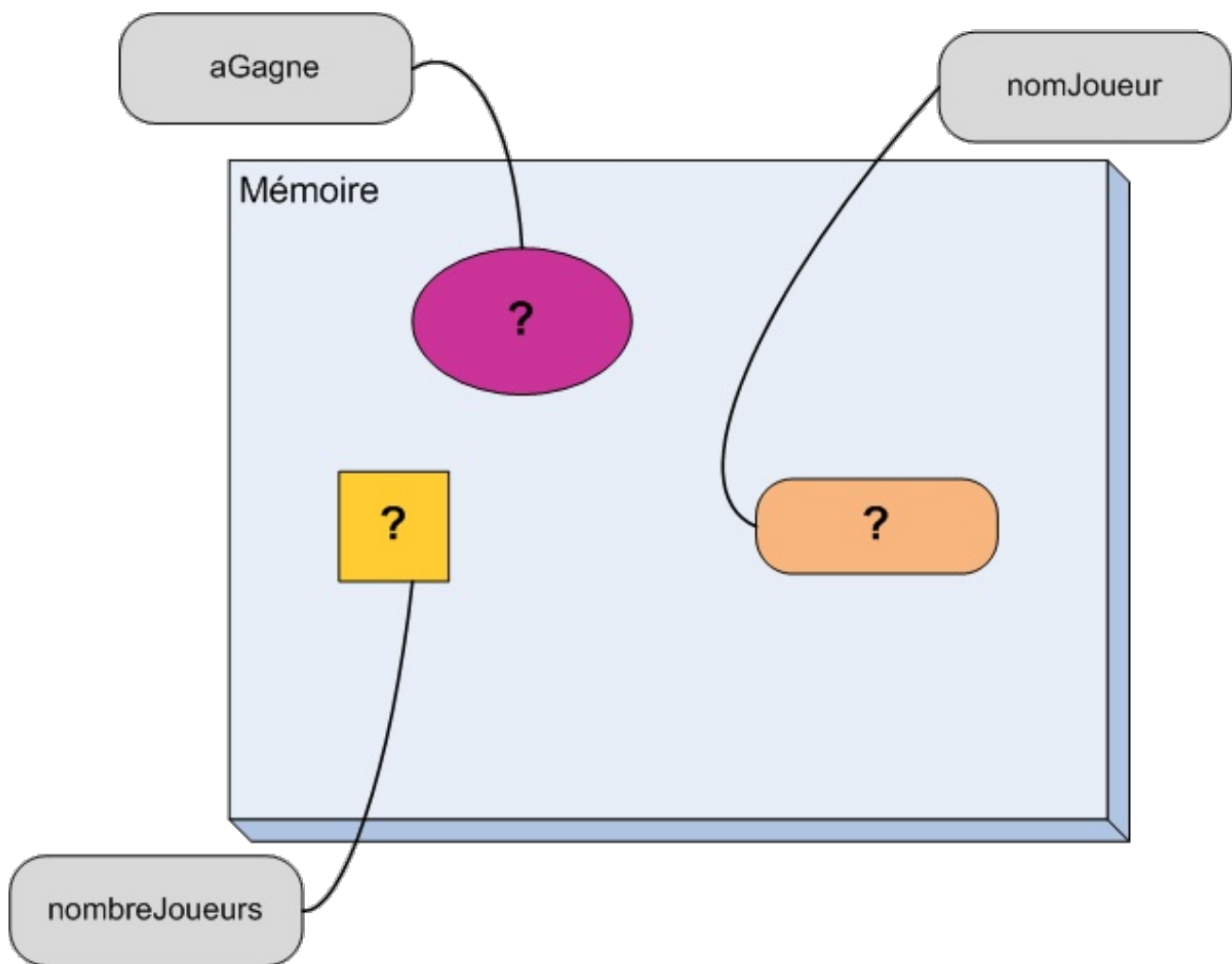
int main()
{
    string nomJoueur;
    int nombreJoueurs;
    bool aGagne;           //Le joueur a-t-il gagné ?

    return 0;
}
```



Une erreur courante est de mettre des parenthèses vides après le nom de la variable. Comme ceci : `int nombreJoueurs ()`. Ceci est incorrect. Il ne faut **pas** mettre de parenthèses, juste le type et le nom.

Simple non ? 😊 Je savais que ça allait vous plaire. Et je vous offre même un schéma en bonus !



On a bien trois cases dans la mémoire et les trois étiquettes correspondantes. La chose nouvelle est que l'on ne sait pas ce que contiennent ces trois cases. Nous verrons dans le chapitre suivant comment modifier le contenu d'une variable et donc remplacer ces points d'interrogation par d'autres valeurs plus intéressantes.



Je viens de vous montrer comment déclarer des variables sans leur donner de valeur initiale. Je vous conseille par contre de **toujours initialiser** vos variables. Ce que je vous ai montré là, n'est à utiliser que dans les cas où l'on ne sait vraiment pas quoi mettre comme valeur. Ce qui est très rare.

Il est temps d'apprendre à effectuer quelques opérations avec nos variables. Parce que vous en conviendrez, pour l'instant, on n'a pas appris grand chose d'utile. Notre écran est resté désespérément vide. 😞

Afficher la valeur d'une variable

Dans le chapitre précédent, vous avez appris à afficher du texte à l'écran. J'espère que vous vous souvenez encore de ce qu'il faut faire.

Oui, c'est bien ça. Il faut utiliser `cout` et les chevrons (`<<`). Parfait. Parce que pour afficher le contenu d'une variable, c'est la même chose. A la place du texte à afficher, on met simplement le nom de la variable.

Code : C++ - Afficher le contenu d'une variable

```
cout << ageUtilisateur;
```

Facile non ?

Prenons un exemple complet pour essayer.

Code : C++ - Exemple d'affichage

```
#include <iostream>
using namespace std;

int main()
{
    int ageUtilisateur(16);
    cout << "Votre age est : ";
    cout << ageUtilisateur;
    return 0;
}
```

Une fois compilé, ce code affiche ceci à l'écran:

Code : Console - Résultat du code précédent

```
Votre age est : 16
```

Exactement ce que l'on voulait ! 😊 On peut même faire encore plus simple. Tout mettre sur une seule ligne ! Et on peut même ajouter un retour à la ligne à la fin.



Pensez à mettre une espace à la fin du texte. Comme ça la valeur de votre variable sera détachée du texte lors de l'affichage.

Code : C++ - Exemple d'affichage

```
#include <iostream>
using namespace std;

int main()
{
    int ageUtilisateur(16);
    cout << "Votre age est : " << ageUtilisateur << endl;
    return 0;
}
```

Et on peut même afficher le contenu de plusieurs variables à la fois.

Code : C++ - Plusieurs variables d'un coup !

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int qiUtilisateur(150);
    string nomUtilisateur("Albert Einstein");

    cout << "Vous vous appelez " << nomUtilisateur << " et votre QI
vaut " << qiUtilisateur << endl;
    return 0;
}
```

Ce qui affiche le résultat escompté.

Code : Console

```
Vous vous appelez Albert Einstein et votre QI vaut 150
```

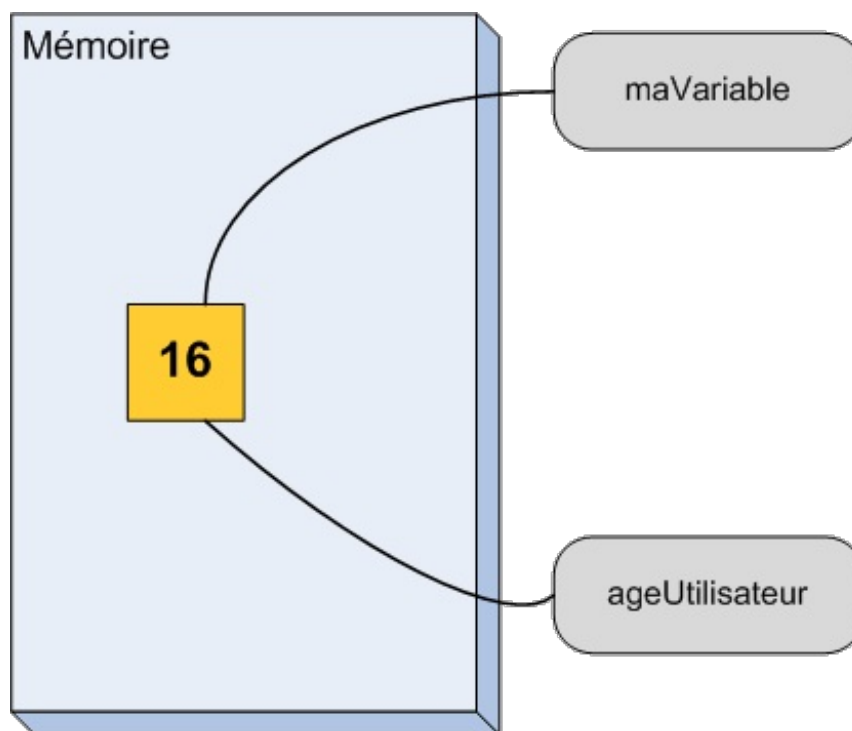
Mais je pense que vous n'en doutiez pas vraiment. Nous verrons dans le chapitre suivant comment faire le contraire, récupérer la saisie d'un utilisateur et la stocker dans une variable.

Les références

Avant de terminer ce chapitre, il nous reste une notion importante à voir. Il s'agit des **références**. Je vous ai expliqué au tout début de ce chapitre qu'une variable pouvait être considérée comme étant une case mémoire avec une étiquette portant son nom. Dans la vraie vie, on peut très bien mettre plusieurs étiquettes sur un objet donné, en C++ c'est la même chose, on peut coller une deuxième (troisième, ..., dixième, etc.) étiquette à une case mémoire.

On obtient alors un deuxième moyen d'accéder à la **même** case mémoire. Un petit peu comme si on donnait un surnom à une variable en plus de son nom normal. On parle parfois d'**alias**, mais le mot correct en C++ est **référence**.

Schématiquement, on peut se représenter une référence comme ceci:



On a une seule case mémoire mais deux étiquettes qui y sont accrochées.

Au niveau du code, on utilise une esperluette (&) pour déclarer une référence sur une variable. Voyons ça avec un petit exemple.

Code : C++ - Déclaration d'une référence

```
int ageUtilisateur(16); //Declaration d'une variable.  
  
int& maVariable(ageUtilisateur); //Declaration d'une reference  
nommee 'maVariable' qui est accrochee a la variable  
'ageUtilisateur'.
```

A la ligne 1, on déclare une case mémoire nommée `ageUtilisateur` dans laquelle on met le nombre **16**. Et à la ligne 3, on accroche une deuxième étiquette à cette case mémoire. On a donc dorénavant deux moyens d'accéder au même espace dans la mémoire de notre ordinateur. 😎

On dit que `maVariable` **fait référence** à `ageUtilisateur`.



La référence doit impérativement être du même type que la variable à laquelle elle est accrochée ! Un `int&` ne peut faire référence qu'à un `int`, de même qu'un `string&` ne peut être associé qu'à une variable de type `string`.

Essayons pour voir. On peut afficher l'âge de l'utilisateur comme d'habitude et via une référence.

Code : C++ - Exemple d'utilisation d'une référence

```
#include <iostream>
using namespace std;

int main()
{
    int ageUtilisateur(18);    //Une variable pour contenir l'âge de
    l'utilisateur
    int& maReference(ageUtilisateur); //Et une référence sur la
    variable ageUtilisateur

    //On peut, a partir d'ici, utiliser 'ageUtilisateur' ou
    'maReference' indistinctement.
    //puisque ce sont deux étiquettes de la même case en mémoire

    cout << "Vous avez " << ageUtilisateur << " ans. (via variable)"
    << endl; //On affiche comme toujours
    cout << "Vous avez " << maReference << " ans. (via reference)"
    << endl;    //Et on affiche en utilisant la référence

    return 0;
}
```

Ce qui donne évidemment le résultat escompté.

Code : Console

```
Vous avez 18 ans. (via variable)
Vous avez 18 ans. (via reference)
```

Une fois qu'elle a été déclarée, on peut manipuler la référence comme si on manipulait la variable elle-même. Il n'y a **aucune différence** entre les deux.



Euh... Mais à quoi est-ce que ça peut bien servir ?

Bonne question ! 😊 C'est vrai que dans l'exemple que je vous ai donné, on peut très bien s'en passer. Mais imaginez que l'on ait besoin de cette variable dans deux parties très différentes du programme, des parties créées par différents programmeurs. Dans une des parties, un des programmeurs va s'occuper de la déclaration de la variable alors que l'autre programmeur va juste l'afficher. Ce deuxième programmeur aura juste besoin d'un accès à la variable et un alias sera donc suffisant.

Pour l'instant, ça vous paraît très abstrait et inutile ? Il faut juste savoir que c'est un des éléments importants du C++ qui apparaîtra à de très nombreuses reprises dans ce cours. Il est donc important de se familiariser avec la notion avant de devoir l'utiliser dans des cas plus compliqués.

Bon !

Récapitulons ce que nous avons appris. A la fin de ce chapitre, vous savez :

- Déclarer une variable, c'est-à-dire demander à l'ordinateur de nous prêter un peu de sa mémoire pour y stocker des informations.

- Afficher le contenu d'une variable à l'écran.
- Définir une référence (un alias) sur une autre variable.

Moi je trouve que c'est pas mal pour un seul chapitre. Il est donc temps de faire une pause avant de continuer avec des choses bien plus intéressantes : recevoir des informations de l'utilisateur et effectuer des opérations. Comme une calculatrice en fait. 😊

Assurez-vous que vous avez bien compris le tout parce que des variables vous allez en manger tout le restant de votre vie et notamment dans le chapitre suivant. 😊

Une vraie calculatrice

J'ai commencé à vous parler de variables dans le chapitre précédent en vous présentant la mémoire d'une calculatrice. Notre ordinateur étant une super-super-super-calculatrice, on doit pouvoir lui faire faire des calculs et pas juste sauvegarder des données. J'espère que ça vous intéresse, parce que c'est ce que je vais vous apprendre à faire.

Nous allons commencer en douceur avec la première tâche qu'on effectue sur une calculette. Vous voyez de quoi je veux parler ? Oui c'est ça, écrire des nombres pour les mettre dans la machine. Nous allons donc voir comment demander des informations à l'utilisateur et comment les stocker dans la mémoire. Nous aurons donc besoin de variables !

Dans un deuxième temps, je vais vous présenter comment effectuer de petits calculs. Finalement, comme vous savez déjà comment afficher un résultat, vous pourrez mettre tout votre savoir en action avec un petit exercice.

Demander des informations à l'utilisateur

Dans le chapitre précédent, je vous ai expliqué comment afficher des variables dans la console. Voyons maintenant comment faire le contraire, c'est-à-dire demander des informations à l'utilisateur pour les stocker dans la mémoire.

Lecture depuis la console

Vous l'aurez remarqué, le C++ utilise pas mal de mots tirés de l'anglais. C'est notamment le cas pour le flux sortant `cout`, qui doit se lire "c-out". Ce qui est bien, c'est qu'on peut immédiatement en déduire le nom du flux entrant. Avec `cout`, les données sortent du programme, d'où le out. Le contraire de out en anglais étant in, qui signifie "vers l'intérieur", on utilise `cin` pour faire entrer des informations dans le programme. `cin` se décompose aussi sous la forme "c-in" et se prononce "si-inne". C'est important pour les soirées entre programmeurs. 😊

Ce n'est pas tout ! Associés à `cout`, il y avait les chevrons (`<<`). Dans le cas de `cin`, il y en a aussi, mais **dans l'autre sens** (`>>`).

Voyons ce que ça donne avec un premier exemple.

Code : C++ - Premier exemple de cin

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Quel age avez-vous ?" << endl;

    int ageUtilisateur(0);    //On prepare une case memoire pour
    stocker un entier.

    cin >> ageUtilisateur;    //On fait entrer un nombre dans cette case.

    cout << "Vous avez " << ageUtilisateur << " ans !" << endl;
    //Et on l'affiche.

    return 0;
}
```

Je vous invite à tester ce programme. Voici ce que ça donne avec mon âge 😊 :

Code : Console - Quel âge avez-vous ?

```
Quel age avez-vous ?
22
Vous avez 22 ans !
```



Que s'est-il passé exactement ?

Le programme a affiché le texte `Quel age avez-vous ?`. Jusque-là, rien de bien sorcier. Puis, comme on l'a vu précédemment, à la ligne 8, le programme demande une case mémoire pour stocker un `int` à l'ordinateur et il baptise cette case `ageUtilisateur`.


Ensuite, ça devient vraiment intéressant. L'ordinateur affiche un curseur blanc clignotant et attend que l'utilisateur écrive quelque chose. Quand celui-ci a terminé et appuyé sur Entrée, le programme prend ce qui a été écrit et met le contenu dans la case mémoire `ageUtilisateur` à la place du 0 qui s'y trouvait.

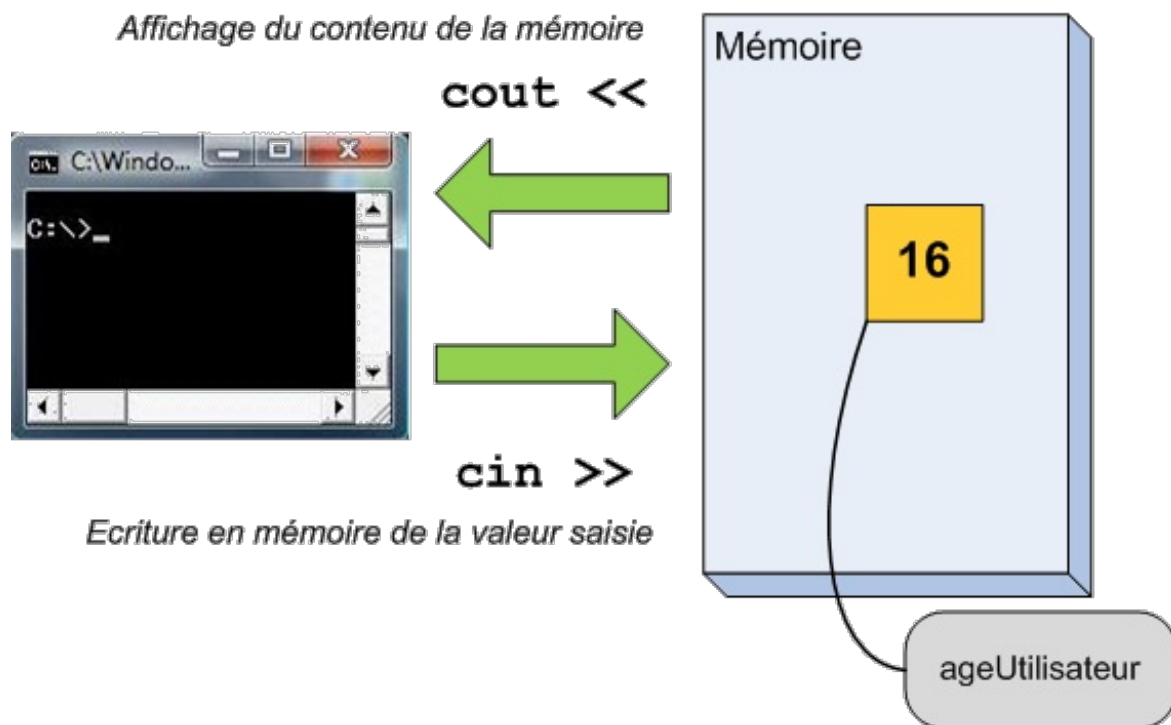
Finalement, on retombe sur quelque chose de connu, puisque le programme affiche une petite phrase et le contenu de la variable.

Une astuce pour les chevrons

Il arrive souvent que l'on se trompe dans le sens des chevrons. Vous ne seriez pas le premier à écrire `cout >>` ou `cin <<`, ce qui est faux.

Pour se souvenir du sens correct, je vous conseille de considérer les chevrons comme si c'étaient des flèches indiquant la direction dans laquelle les données se déplacent. Depuis la variable vers `cout` ou depuis `cin` vers votre variable.

Le mieux est de prendre un petit schéma magique. 



Quand on affiche la valeur d'une variable, les données sortent du programme, on utilise donc une flèche allant de la variable vers `cout`. Quand on demande une information à l'utilisateur, c'est le contraire, la valeur vient de `cin` et va dans la variable.

Avec ça, plus moyen de se tromper !

D'autres variables

Évidemment, ce que je vous ai présenté marche aussi avec d'autres types de variables. Voyons ça avec un petit exemple.

Code : C++ - Lecture d'autres types de variables

```
#include <iostream>
#include <string>
using namespace std;
```



```

int main()
{
    cout << "Quel est votre prenom ?" << endl;
    string nomUtilisateur("Sans nom");           //On crée une case
    mémoire pour contenir une chaîne de caractères
    cin >> nomUtilisateur;                       //On remplit cette
    case avec ce qu'écrit l'utilisateur

    cout << "Combien vaut pi ?" << endl;
    double piUtilisateur(-1.);                   //On crée une case
    mémoire pour stocker un nombre réel
    cin >> piUtilisateur;                         //Et on remplit
    cette case avec ce qu'écrit l'utilisateur

    cout << "Vous vous appelez " << nomUtilisateur << " et vous
    pensez que pi vaut " << piUtilisateur << "." << endl;

    return 0;
}

```

Je crois que je n'ai même pas besoin de donner d'explications. Je vous invite néanmoins à tester pour bien comprendre en détail ce qui se passe.

Le problème des espaces

Avez-vous testé le code précédent en mettant votre nom et prénom ? Regardons ce que ça donne.

Code : Console - Les soucis d'Albert

```

Quel est votre prenom ?
Albert Einstein
Combien vaut pi ?
Vous vous appelez Albert et vous pensez que pi vaut 0.

```



L'ordinateur n'a rien demandé pour pi et le nom de famille a disparu ! Que s'est-il passé ?

C'est un problème d'espaces. Quand on appuie sur Entrée, l'ordinateur copie ce qui a été écrit par l'utilisateur dans la case mémoire. Mais, il s'arrête à la première *espace* ou au premier *retour à la ligne*. Quand il s'agit d'un nombre, cela ne pose pas de problèmes puisqu'il n'y a pas d'espaces dans les nombres.

Pour les `string`, le problème se pose. Il peut très bien y avoir une espace dans une chaîne de caractère. Et donc l'ordinateur va couper au mauvais endroit, c'est-à-dire après le premier mot. Et comme il n'est pas très malin, il va croire que le nom de famille correspond à la valeur de pi !

Il faudrait en fait pouvoir récupérer toute la ligne plutôt que juste le premier mot. Et si je vous le propose, c'est qu'il y a une solution pour le faire !

Il faut utiliser la **fonction** `getline()`. Nous verrons plus loin ce que sont exactement les fonctions, mais pour l'instant voyons comment faire dans ce cas particulier.

Il faut remplacer la ligne `cin >> nomUtilisateur;` par un `getline()`.

Code : C++ - Lecture d'une ligne complète

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    cout << "Quel est votre nom ?" << endl;

```

```

    string nomUtilisateur("Sans nom");           //On crée une case
    mémoire pour contenir une chaîne de caractères
    getline(cin, nomUtilisateur); //On remplit cette case avec toute la
    ligne que l'utilisateur a écrite

    cout << "Combien vaut pi ?" << endl;
    double piUtilisateur(-1.);                   //On crée une case
    mémoire pour stocker un nombre réel
    cin >> piUtilisateur;                         //Et on remplit
    cette case avec ce qu'écrit l'utilisateur

    cout << "Vous vous appelez " << nomUtilisateur << " et vous
    pensez que pi vaut " << piUtilisateur << "." << endl;

    return 0;
}

```

On retrouve les mêmes éléments qu'auparavant. Il y a `cin` et il y a le nom de la variable (`nomUtilisateur`), sauf que cette fois, le tout se trouve entre des parenthèses et séparé par une virgule et pas par des chevrons.



L'ordre des éléments entre les parenthèses est très important. Il faut absolument mettre le `cin` en premier !

Cette fois le nom ne sera pas tronqué lors de la lecture et notre ami Albert pourra utiliser notre programme sans soucis. 😊

Code : Console - Plus de soucis pour Albert

```

Quel est votre nom ?
Albert Einstein
Combien vaut pi ?
3.14
Vous vous appelez Albert Einstein et vous pensez que pi vaut 3.14.

```

Demander d'abord la valeur de pi

Si l'on utilise d'abord `cin >>` puis `getline()`, par exemple en demandant d'abord la valeur de π avant de demander le nom, le code ne marche pas. L'ordinateur ne demande pas son nom à l'utilisateur et affiche n'importe quoi. Pour palier ce problème, il faut ajouter la ligne `cin.ignore()` après l'utilisation des chevrons.

Code : C++

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    cout << "Combien vaut pi ?" << endl;
    double piUtilisateur(-1.);           //On crée une case
    mémoire pour stocker un nombre réel
    cin >> piUtilisateur;                 //Et on remplit
    cette case avec ce qu'écrit l'utilisateur

    cin.ignore();

    cout << "Quel est votre nom ?" << endl;
    string nomUtilisateur("Sans nom");   //On crée une case
    mémoire pour contenir une chaîne de caractères
    getline(cin, nomUtilisateur);        //On remplit cette

```

```

    case avec toute la ligne que l'utilisateur a écrite

    cout << "Vous vous appelez " << nomUtilisateur << " et vous
    pensez que pi vaut " << piUtilisateur << "." << endl;

    return 0;
}

```

Avec ça, plus de soucis. 😊

Quand on mélange l'utilisation des chevrons et de `getline()`, il faut toujours faire `cin.ignore()` après la ligne `cin>>a`. C'est une règle à apprendre. 😊

Voyons maintenant ce que l'on peut faire avec des variables. Par exemple, additionner deux nombres.

Modifier des variables

Changer le contenu d'une variable

Je vous ai expliqué dans l'introduction de ce chapitre que la mémoire de l'ordinateur ressemblait dans sa manière de fonctionner à celle d'une calculatrice. Ce n'est pas la seule similitude. On peut évidemment effectuer des opérations sur un ordinateur. Et cela se fait en utilisant des variables.

Commençons par voir comment changer le contenu d'une variable. On utilise le symbole `=` pour effectuer un changement de valeur. Si j'ai une variable de type `int` dont je veux modifier le contenu, j'écris le nom de ma variable, suivi d'un `=` et finalement la nouvelle valeur. C'est ce qu'on appelle **l'affectation d'une variable**.

Code : C++ - Affectation d'une valeur

```

int unNombre(0); //Je crée une case mémoire nommée 'unNombre' et
qui contient le nombre 0.

unNombre = 5; //Je mets 5 dans la case mémoire 'unNombre'.

```

On peut aussi directement affecter le contenu d'une variable à une autre.

Code : C++ - Affectation d'une variable à une autre

```

int a(4), b(5); //Déclaration de deux variables.

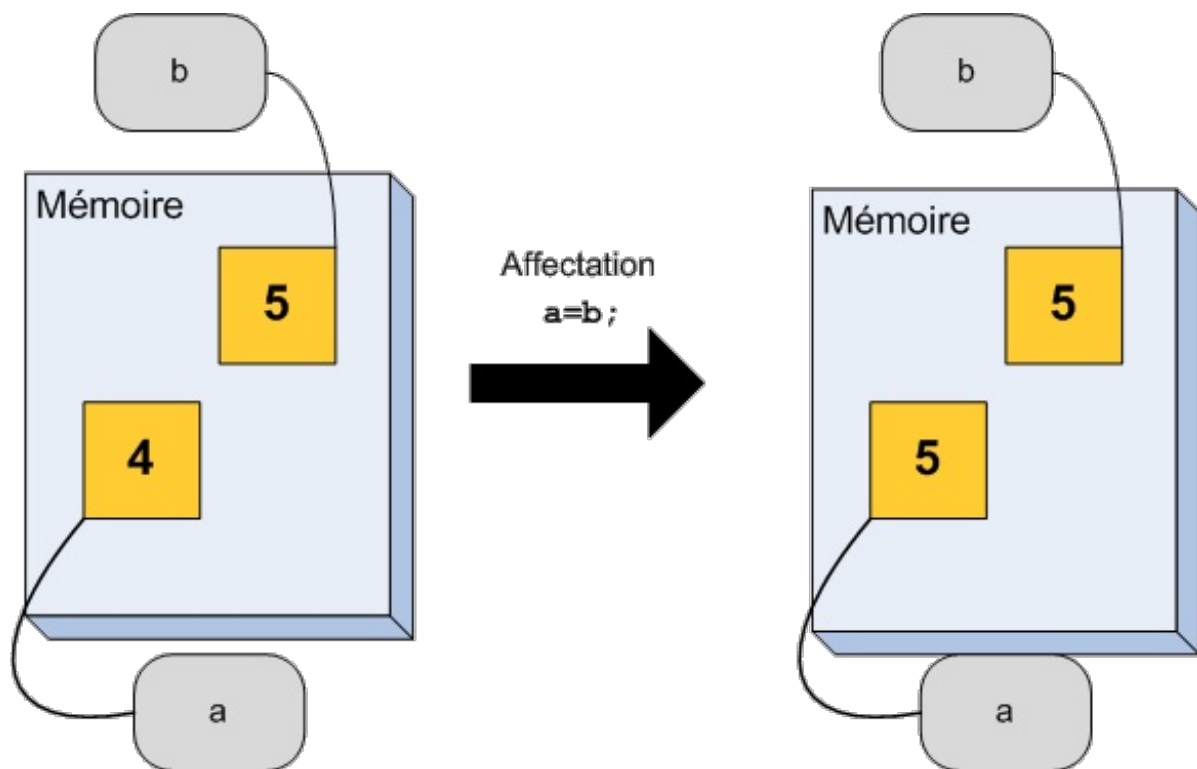
a = b; //Affectation de la valeur de 'b' à 'a'.

```



Que se passe-t-il exactement ?

Quand il arrive à la ligne 3 du code précédent, l'ordinateur va lire le contenu de la case mémoire nommée `b`, soit le nombre **5**. Il va ensuite ouvrir la case dont le nom est `a` et il y écrit la valeur **5** en remplaçant le **4** qui s'y trouvait. Voyons ça avec un schéma.



On peut d'ailleurs afficher le contenu des deux variables pour vérifier.

Code : C++ - Test de l'affectation d'une variable à une autre

```
#include <iostream>
using namespace std;

int main()
{
    int a(4), b(5); //Déclaration de deux variables.

    cout << "a vaut : " << a << " et b vaut : " << b << endl;

    cout << "Affectation !" << endl;
    a = b; //Affectation de la valeur de 'b' à 'a'.

    cout << "a vaut : " << a << " et b vaut : " << b << endl;

    return 0;
}
```

Avez-vous testé ? Non ? N'oubliez pas qu'il est important de tester les codes proposés pour bien comprendre. Bon, comme je suis gentil, je vous donne le résultat.

Code : Console

```
a vaut : 4 et b vaut : 5
Affectation !
a vaut : 5 et b vaut : 5
```

Exactement ce que je vous avais prédit.



La valeur de **b** n'a pas changé ! Il est important de se rappeler que lors d'une affectation, seule la variable **à gauche** du symbole **=** est modifiée.



Cela ne veut pas dire que les deux variables sont égales ! Juste que le contenu de celle de droite est copié dans celle de gauche.

C'est un bon début, mais on est encore loin d'une calculatrice. Il nous manque ...

... les opérations !

Une vraie calculatrice de base !

Commençons avec l'opération la plus simple, l'addition bien sûr. Et je pense que je ne vais pas trop vous surprendre en vous disant qu'on utilise le symbole +.

C'est vraiment très simple à faire :

Code : C++ - Votre première addition

```
int a(5), b(8), resultat(0);  
resultat = a + b; //Et hop une addition pour la route!
```

Comme c'est votre première opération, je vous décris ce qui se passe précisément. A la ligne 1, le programme crée trois cases dans la mémoire, nommées a, b et resultat. Il remplit également ces cases avec les valeurs **5**, **8** et **0** respectivement. Tout ça, on commence à connaître. 😊

On arrive ensuite à la ligne 3. L'ordinateur voit qu'il va devoir modifier le contenu de la variable resultat. Il regarde alors ce qu'il y a de l'autre côté du = et il trouve qu'il va devoir faire la somme du contenu de ce qui se trouve dans les cases mémoire a et b. Il regarde alors le contenu de a et de b **sans le modifier**, effectue le calcul et écrit la somme dans la variable resultat. Tout ça en un éclair. Pour calculer, l'ordinateur est un vrai champion.

On peut même vérifier que ça fonctionne si vous voulez.

Code : C++ - Vérification

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int resultat(0), a(5), b(8);  
  
    resultat = a + b;  
  
    cout << "5 + 8 = " << resultat << endl;  
    return 0;  
}
```

Et sur votre écran vous devriez voir :

Code : Console

```
5 + 8 = 13
```

Et ce n'est pas tout, il existe encore quatre autres opérations. Je vous ai mis un résumé des possibilités dans un petit tableau récapitulatif.

Opération	Symbole	Exemple
-----------	---------	---------

Addition	+	resultat = a + b;
Soustraction	-	resultat = a - b;
Multiplication	*	resultat = a * b;
Division	/	resultat = a / b;
Modulo	%	resultat = a % b;



Mais, qu'est-ce que le modulo ? Je n'ai pas vu ça à l'école.

Je suis sûr que si, mais pas forcément sous ce nom là. Il s'agit en fait du reste de la division entière. Par exemple, si vous ressortez vos cahiers d'école vous devriez retrouver des calculs tels que **13** divisé par **3**.

Comme **13** n'est pas un multiple de **3**, il faut déduire quelque chose pour en obtenir un. C'est ce "quelque chose" qu'on appelle le reste de la division. Avec notre exemple, on peut écrire **13 = 4 * 3 + 1**. L'opérateur modulo calcule ce reste de la division.

Peut-être que vous en aurez besoin un jour. 😊



Cet opérateur n'existe que pour les nombres entiers !

A partir des opérations de base, on peut tout à fait écrire des expressions mathématiques plus complexes qui nécessitent plusieurs variables. On peut également utiliser des parenthèses si nécessaire.

Code : C++

```
int a(2), b(4), c(5), d; //Quelques variables
d = ((a+b) * c) - c; //Un calcul compliqué !
```

La seule limite est votre imagination. Toute expression valide en math, l'est aussi en C++. 🧙

Les constantes

Je vous ai présenté comment modifier des variables. J'espère que vous avez bien compris ! Parce qu'on va faire le contraire, en quelque sorte. Je vais vous montrer comment déclarer des variables non modifiables. 😊

En termes techniques, on parle de **constantes**. Ça fait beaucoup de termes techniques pour un seul chapitre, mais je vous promets que dans la suite, ça va se calmer. 😊



Euh, mais à quoi ça peut bien servir des variables non modifiables ?

Ah, je savais que vous alliez poser cette question. Je vous ai donc préparé une réponse aux petits oignons.

Prenons le futur jeu vidéo révolutionnaire que vous allez créer. Comme vous êtes très fort, je pense qu'il y aura plusieurs niveaux, disons 10. Et bien ce nombre de niveaux ne va jamais changer durant l'exécution du programme. Entre le moment où l'utilisateur a lancé le jeu et le moment où il l'a quitté, il y a eu en permanence 10 niveaux dans votre jeu. Ce nombre est constant. En C++, on pourrait donc créer une variable `nombreNiveaux` qui serait une constante.

Ce n'est bien sûr pas le seul exemple. Pensez à une calculatrice, elle aura besoin de la constante π ou bien à un jeu où les personnages tombent, il faudra utiliser la constante d'accélération de la pesanteur $g = 9.81$, etc.

Ces valeurs ne vont jamais changer. π vaudra toujours **3.14** et l'accélération sur Terre est partout identique. Ce sont des constantes. 😊 On peut même ajouter que ce sont des constantes dont on connaît la valeur lors de la rédaction du code source.

Mais ce n'est pas tout. Il existe aussi des variables dont la valeur ne change jamais mais dont on ne connaît pas la valeur à l'avance. 😊 Prenons le résultat d'une opération dans une calculatrice. Une fois que le calcul est effectué, le résultat ne change

plus. La variable qui contient le résultat est donc une constante.

Voyons donc comment déclarer une telle variable.

Déclarer une constante

C'est très simple. On déclare une variable normale et on ajoute le mot-clé **const** entre le type et le nom.

Code : C++ - Une constante

```
int const nombreNiveaux(10);
```

Et ça marche bien sûr avec tous les types de variables.

Code : C++

```
string const motDePasse("wAsTZsaswQ"); //Le mot de passe secret  
double const pi(3.14);  
unsigned int const pointsDeVieMaximum(100); //Le nombre maximal de  
points de vie
```

Je pourrais encore continuer longtemps, mais je pense que vous avez saisi le principe. Vous n'êtes pas des futurs génies de l'informatique pour rien. 😊



Déclarez toujours **const** tout ce qui est possible. Cela permet d'éviter des erreurs d'inattention lorsque l'on programme mais peut aussi aider le compilateur à créer un programme plus efficace.

Vous verrez, on reparlera des constantes dans les chapitres suivants. En attendant, préparez-vous pour votre premier exercice.



Un premier exercice

Je crois qu'on a enfin toutes les clés en main pour réaliser votre premier vrai programme. Dans l'exemple précédent, le programme effectuait l'addition de deux nombres fixés à l'avance. Il serait bien mieux de demander à l'utilisateur quels nombres il veut additionner ! Voilà donc le sujet de notre premier exercice. Demander deux nombres à l'utilisateur, calculer la somme de ces deux nombres et finalement afficher le résultat. 😊

Rassurez-vous, je vais vous aider, mais je vous invite à essayer par vous-même avant de regarder la solution. C'est le meilleur moyen d'apprendre.

Dans un premier temps, il faut toujours réfléchir aux variables qu'il va falloir utiliser dans le code.



De quoi avons-nous besoin ici ?

Il nous faut une variable pour stocker le premier nombre entré par l'utilisateur et une autre pour stocker le deuxième. En se basant sur l'exemple précédent, on peut simplement appeler ces deux cases mémoires a et b.

Finalement, il faut se poser la question du **type** de nos variables. Nous voulons faire des calculs, il nous faut donc prendre **int**, **unsigned int** ou **double** selon les nombres que l'on veut utiliser. Je vote pour **double**, afin de pouvoir utiliser des nombres à virgule.

On peut donc déjà écrire un bout de notre programme, c'est-à-dire la structure de base et la déclaration des variables.

Code : C++

```
#include <iostream>  
using namespace std;
```

```
int main()
{
    double a(0), b(0); //Déclaration des variables utiles

    //...
    return 0;
}
```

La prochaine étape consiste à demander des nombres à l'utilisateur. Je pense que vous vous en souvenez encore, cela se fait grâce à `cin >>`. On peut donc aller plus loin et écrire :

Code : C++

```
#include <iostream>
using namespace std;

int main()
{
    double a(0), b(0); //Déclaration des variables utiles

    cout << "Bienvenue dans le programme d'addition a+b !" << endl;

    cout << "Donnez une valeur pour a : ";    //Demande du premier
nombre
    cin >> a;

    cout << "Donnez une valeur pour b : ";    //Demande du deuxième
nombre
    cin >> b;

    //...

    return 0;
}
```

Il ne nous reste plus qu'à effectuer l'addition et afficher le résultat. Il nous faut donc une variable. Comme le résultat du calcul ne va pas changer, nous pouvons (et même devons) déclarer cette variable comme une constante. Nous allons remplir cette constante, que j'appelle `resultat`, avec le résultat du calcul. Finalement, pour effectuer l'addition, c'est bien sûr le `+` qu'il va falloir écrire.

Code : C++ - Correction du premier exercice

```
#include <iostream>
using namespace std;

int main()
{
    double a(0), b(0); //Déclaration des variables utiles

    cout << "Bienvenue dans le programme d'addition a+b !" << endl;

    cout << "Donnez une valeur pour a : ";    //Demande du premier
nombre
    cin >> a;

    cout << "Donnez une valeur pour b : ";    //Demande du deuxième
nombre
    cin >> b;

    double const resultat(a + b);    //On effectue l'opération

    cout << a << " + " << b << " = " << resultat << endl; //On
affiche le résultat
}
```

```
    return 0;  
}
```

Mmmh, ça a l'air rudement bien tout ça ! Compilons et testons pour voir.

Code : Console

```
Bienvenue dans le programme d'addition a+b !  
Donnez une valeur pour a : 123.784  
Donnez une valeur pour b : 51.765  
123.784 + 51.765 = 175.549
```

Magnifique ! Exactement ce qui était prévu !

Bon, j'ai assez travaillé. A vous maintenant de programmer. Je vous propose de vous entrainer en modifiant cet exercice. Voici quelques idées:

- Calculer le produit de a et b plutôt que leur somme.
- Faire une opération plus complexe comme $a * b + c$.
- Demander deux nombres entiers et calculer leur quotient et le reste de la division.

Bon courage et amusez-vous bien !

Les raccourcis

Après cet exercice, vous savez manipuler toutes les opérations de base. C'est peut-être surprenant pour vous, mais il n'en existe pas d'autres ! 😊 Avec ces 5 opérations, on peut tout faire, même des jeux vidéo comme ceux présentés dans le chapitre d'introduction.

Il existe quand même quelques variantes qui, j'en suis sûr, vont vous plaire.

L'incrémentation

Une des opérations les plus courantes en informatique c'est ajouter 1 à une variable. Pensez par exemple aux cas suivants :

- Passer du niveau 4 au niveau 5 de votre jeu.
- Augmenter le nombre de vie du personnage.
- Ajouter un joueur à la partie.
- etc.

Cette opération est tellement courante qu'elle a un nom spécial. On parle d'**incrémentation**. Avec vos connaissances actuelles, vous savez déjà comment incrémenter une variable.

Code : C++

```
int nombreJoueur(4); //Il y a 4 joueurs dans la partie  
nombreJoueur = nombreJoueur + 1; //On en ajoute un  
//A partir d'ici, il y a 5 joueurs
```

Bien ! Mais comme je vous l'ai dit, les informaticiens sont des fainéants et la 2^e ligne de ce code est un peu trop longue à écrire. Les créateurs du C++ ont donc inventé une notation spéciale pour ajouter 1 à une variable. Voici comment.

Code : C++

```
int nombreJoueur(4); //Il y a 4 joueurs dans la partie
```

```
++nombreJoueur;  
//A partir d'ici, il y a 5 joueurs
```

On utilise le symbole ++. On écrit ++ suivi du nom de la variable et finalement on met le point-virgule habituel. Ce code a **exactement le même effet** que le précédent. Il est juste plus court à écrire.

On peut également écrire `nombreJoueur++`. C'est-à-dire, placer l'opérateur ++ après le nom de la variable. On appelle ça la post-incrémentation à l'opposé de la pré-incrémentation lorsque l'on place le ++ avant le nom de la variable.

Vous trouvez peut-être ça ridicule, mais je suis sûr que vous allez rapidement adorer ce genre de choses ! 😊



Cette astuce est tellement utilisée qu'elle est même présente dans le nom du langage ! Oui, oui, C++ veut en quelque sorte dire "C incrémenté", ou en meilleur français, "C amélioré". Ils sont fous ces informaticiens. 🤪

La décrémentation

La **décrémentation** est l'opération inverse. Soustraire 1 à une variable.

La version sans raccourci s'écrit comme ceci.

Code : C++

```
int nombreJoueur(4); //Il y a 4 joueurs dans la partie  
nombreJoueur = nombreJoueur - 1; //On en enlève un  
//A partir d'ici, il y a 3 joueurs
```

Je suis presque sûr que vous connaissez la version courte. On utilise ++ pour ajouter 1, c'est donc -- qu'il faut utiliser pour soustraire 1.

Code : C++

```
int nombreJoueur(4); //Il y a 4 joueurs dans la partie  
--nombreJoueur; //On en enlève un  
//A partir d'ici, il y a 3 joueurs
```

A nouveau, la syntaxe inversée est possible. On peut donc écrire `nombreJoueur--` à la place de `--nombreJoueur`.

Simple, non ? 😎

Les autres opérations

Bon. Ajouter ou enlever 1, c'est bien, mais c'est pas non plus suffisant pour tout faire. Il existe des raccourcis pour toutes les opérations de base.

Si l'on souhaite diviser une variable par 3, on devrait écrire en version longue :

Code : C++

```
double nombre(456);  
nombre = nombre / 3;  
//A partir d'ici, nombre vaut 456/3 = 152
```

La version courte utilise le symbole /= pour obtenir **exactement le même** résultat.

Code : C++

```
double nombre(456);  
nombre /= 3;  
//A partir d'ici, nombre vaut 456/3 = 152
```

Il existe des raccourcis pour les 5 opérations de base, c'est-à-dire, +=, -=, *=, /= et % =.

Je suis sûr que vous n'allez plus pouvoir vous en passer. Les essayer, c'est les adopter. 😊

Je vous propose un petit exemple pour la route.

Code : C++ - Utilisation des versions raccourcies des opérateurs

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    double nombre(5.3);  
    nombre += 4.2;      //nombre vaut maintenant 9.5  
    nombre *= 2.;       //nombre vaut maintenant 19  
    nombre -= 1.;       //nombre vaut maintenant 18  
    nombre /= 3.;       //nombre vaut maintenant 6  
    return 0;  
}
```

Ces opérations sont utiles quand il faut ajouter ou soustraire autre chose que 1.

Encore plus de maths !

Vous en voulez encore ? Ah je vois, vous n'êtes pas satisfait de votre calculatrice. C'est vrai qu'elle est encore un peu pauvre, elle ne connaît que les opérations de base. Pas vraiment génial pour la super-super-calculatrice qu'est votre ordinateur.

Ne partez pas ! J'ai mieux à vous proposer.

L'en-tête cmath

Pour avoir accès à plus de fonctions mathématiques, il faut ajouter une ligne en haut de votre programme, comme lorsque l'on désire utiliser des variables de type `string`. Il faut ajouter

Code : C++ - Ligne à ajouter pour les fonctions mathématiques

```
#include <cmath>
```

Jusque là, c'est très simple. Et dans `cmath` il y a "math", ce qui devrait vous réjouir. On est sur la bonne voie. 😊



Je vais vous présenter comment utiliser quelques fonctions mathématiques en C++. Il se peut très bien que vous ne sachiez pas ce que sont ces fonctions. Ce n'est pas grave, elles ne vous seront pas utiles dans la suite du cours. Vous saurez ce qu'elles représentent quand vous aurez fait un peu plus de maths.

Commençons avec une fonction très souvent utilisée, la racine carrée. En anglais, racine carrée se dit *square root* et en abrégé on

écrit parfois **sqrt**. 🤖 Comme le C++ utilise l'anglais, c'est ce mot là qu'il va falloir retenir et utiliser.

Pour utiliser une fonction mathématique, on écrit le nom de la fonction, suivi de la valeur à calculer entre parenthèses. On utilise alors l'affectation pour stocker le résultat dans une variable.

Code : C++

```
resultat = fonction(valeur);
```

C'est comme en math quand on écrit $y = f(x)$. Il faut juste se souvenir du nom compliqué des fonctions. Pour la racine carrée, cela donnerait `resultat = sqrt(valeur);`.



N'oubliez pas le point-virgule à la fin de la ligne !

Prenons un exemple complet, je pense que vous allez comprendre rapidement le principe.

Code : C++ - La racine carrée

```
#include <iostream>
#include <cmath> //Ne pas oublier cette ligne
using namespace std;

int main()
{
    double const nombre(16);           //Le nombre dont on veut la
    racine                               //Comme sa valeur ne changera
    pas on met 'const'                  //Une case mémoire pour stocker
    double resultat;                    le résultat

    resultat = sqrt(nombre); //On effectue le calcul !

    cout << "La racine de " << nombre << " est " << resultat <<
endl;

    return 0;
}
```

Voyons ce que ça donne.

Code : Console

```
La racine de 16 est 4
```

Wow ! Votre ordinateur calcule correctement. Mais je ne pense pas que vous en doutiez. 🤖

Voyons s'il y a d'autres fonctions à disposition.

Quelques autres fonctions présentes dans cmath

Comme il y a beaucoup de fonctions, je vous propose de tout mettre dans un tableau.

Nom de la fonction	Symbole mathématique	Nom de la fonction en C++	Mini-exemple
Racine carrée	\sqrt{x}	<code>sqrt()</code>	<code>resultat = sqrt(valeur);</code>
Sinus	$\sin(x)$	<code>sin()</code>	<code>resultat = sin(valeur);</code>
Cosinus	$\cos(x)$	<code>cos()</code>	<code>resultat = cos(valeur);</code>
Tangente	$\tan(x)$	<code>tan()</code>	<code>resultat = tan(valeur);</code>
Exponentielle	e^x	<code>exp()</code>	<code>resultat = exp(valeur);</code>
Logarithme népérien	$\ln x$	<code>log()</code>	<code>resultat = log(valeur);</code>
Logarithme en base 10	$\log_{10} x$	<code>log10()</code>	<code>resultat = log10(valeur);</code>
Valeur absolue	$ x $	<code>fabs()</code>	<code>resultat = fabs(valeur);</code>
Arrondi vers le bas	$\lfloor x \rfloor$	<code>floor()</code>	<code>resultat = floor(valeur);</code>
Arrondi vers le haut	$\lceil x \rceil$	<code>ceil()</code>	<code>resultat = ceil(valeur);</code>



Les fonctions trigonométriques (`sin()`, `cos()`, ...) utilisent les radians comme unité d'angle.

Il existe encore beaucoup d'autres fonctions ! Je ne vous ai mis que les principales pour pas qu'on se perde.



On parle de mathématiques, ces fonctions ne sont donc utilisables qu'avec des variables qui représentent des nombres (`double`, `int` et `unsigned int`). Prendre la racine carrée d'une lettre ou calculer le cosinus d'une phrase n'ont de toute façon pas de sens. 😬

Le cas de la fonction puissance

Comme toujours, il y a un cas particulier : la fonction puissance. Comment calculer 4^5 ? Il faut utiliser la fonction `pow()` qui est un peu spéciale. Elle prend **deux arguments**, c'est-à-dire qu'il faut lui mettre deux valeurs entre les parenthèses. Comme pour la fonction `getline()` dont je vous ai parlé avant.

Si je veux calculer 4^5 , je vais devoir faire comme ceci.

Code : C++

```
double const a(4);
double const b(5);
double const resultat = pow(a,b);
```

Je déclare une variable (constante) pour mettre le **4**, une autre pour mettre le **5** et finalement une dernière pour le résultat. Rien de nouveau jusque là. J'utilise la fonction `pow()` pour effectuer le calcul et j'utilise le symbole `=` pour initialiser la variable `resultat` avec la valeur calculée par la fonction.

Nous pouvons donc reprendre l'exercice précédent et remplacer l'addition par notre nouvelle amie, la fonction puissance. 😬 Je vous laisse essayer.

Voici ma version:

Code : C++ - La puissance de `pow()`

```
#include <iostream>
```

```

#include <cmath> //Ne pas oublier !
using namespace std;

int main()
{
    double a(0), b(0); //Déclaration des variables utiles

    cout << "Bienvenue dans le programme de calcul de a^b !" << endl;

    cout << "Donnez une valeur pour a : ";    //Demande du premier
    nombre
    cin >> a;

    cout << "Donnez une valeur pour b : ";    //Demande du deuxième
    nombre
    cin >> b;

    double const resultat(pow(a, b));    //On effectue l'opération
    //On peut aussi écrire comme avant :
    //double const resultat = pow(a,b);
    //Souvenez-vous des deux formes possibles de l'initialisation
    d'une variable

    cout << a << " ^ " << b << " = " << resultat << endl; //On
    affiche le résultat

    return 0;
}

```

Vous avez fait la même chose ? Parfait ! 😊 Vous êtes un futur champion du C++ ! Voyons quand même ce que ça donne.

Code : Console

```

Bienvenue dans le programme de calcul de a^b !
Donnez une valeur pour a : 4
Donnez une valeur pour b : 5
4 ^ 5 = 1024

```

J'espère que vous êtes satisfaits avec toutes ces fonctions mathématiques. Je ne sais pas si vous en aurez besoin un jour. Si c'est le cas, vous saurez où en trouver une description. 😊

Nous voilà au terme de deux gros chapitres sur la mémoire. Dans cette deuxième partie, vous avez appris à modifier les variables grâce au symbole = et à effectuer des opérations mathématiques.

Dans le chapitre suivant, nous allons laisser un peu la mémoire de côté pour nous intéresser à des moyens de modifier le déroulement d'un programme. Une vraie aventure !

Les structures de contrôle

Les programmes doivent être capables de prendre des décisions. Pour y parvenir, les développeurs utilisent ce qu'on appelle des **structures de contrôle**. Ce nom un peu barbare cache en fait deux éléments que nous verrons dans ce chapitre :

- Les conditions : elles permettent d'écrire dans le programme des règles comme "Si ceci arrive, alors fais cela".
- Les boucles : elles permettent de répéter une série d'instructions plusieurs fois.

Savoir manier les structures de contrôle est fondamental ! Bien que ce chapitre ne soit pas réellement difficile, il faut faire attention à bien comprendre ces notions de base. Elles vous serviront durant toute votre vie de développeurs C++... mais aussi dans d'autres langages, car le principe y est le même !



Ces outils sont en fait exactement les mêmes en C++ qu'en C. Si vous avez déjà programmé en C, vous ne devriez donc pas être dépayés.

Et si vous n'avez jamais fait de C de votre vie, ce n'est pas grave, nous allons tout apprendre dans ce chapitre. 😊

Les conditions

Pour qu'un programme soit capable de prendre des décisions, on utilise des conditions dans le code source (on parle aussi de "structures conditionnelles"). Le principe est simple : vous voulez que votre programme réagisse différemment en fonction des circonstances. Nous allons découvrir ici comment utiliser ces fameuses conditions dans nos programmes C++. 😊

Pour commencer, il faut savoir que les conditions permettent de tester des variables. Vous vous souvenez de ces variables stockées en mémoire que nous avons découvertes ? Eh bien nous allons maintenant apprendre à les analyser : "Est-ce que cette variable est supérieure à 10 ?", "Est-ce que cette variable contient bien le mot de passe secret ?"...

Pour effectuer ces tests, nous utilisons des symboles. Voici le tableau des symboles à connaître par coeur :

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de



Faites très attention, il y a bien 2 symboles "=" pour tester l'égalité. Les débutants oublient souvent cela et n'écrivent qu'un seul "=", ce qui n'a pas la même signification en C++.

Nous allons utiliser ces symboles pour effectuer des comparaisons dans nos conditions.

Il faut savoir qu'il existe plusieurs types de conditions en C++ pour faire des tests, mais la plus importante qu'il faut impérativement connaître est sans aucun doute la condition `if`.

La condition `if`

Comme je vous le disais, les conditions permettent de tester des variables. Je vous propose donc de créer un petit programme en même temps que moi et de faire des tests pour vérifier que vous avez bien compris le principe.

On va commencer avec ce code :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(2);

    return 0;
}
```

Ce code-là ne fait rien pour le moment. Il se contente de déclarer une variable nbEnfants (qui indique un nombre d'enfants donc), puis il s'arrête.

Une première condition if

Imaginons qu'on souhaite afficher un message de félicitations si la personne a des enfants. On va ajouter une condition qui regarde si le nombre d'enfants est supérieur à 0 et qui affiche un message dans ce cas.

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(2);

    if (nbEnfants > 0)
    {
        cout << "Vous avez des enfants, bravo !" << endl;
    }

    cout << "Fin du programme" << endl;
    return 0;
}
```

Ce code affiche :

Code : Console

```
Vous avez des enfants, bravo !
Fin du programme
```

Regardez bien la ligne suivante :

```
if (nbEnfants > 0)
```

Elle effectue le test : "Si le nombre d'enfants est supérieur à 0" (if, en anglais, veut dire "si").

Si ce test est vérifié (donc si la personne a bien des enfants), alors l'ordinateur va lire les lignes qui se trouvent entre les accolades : il va donc afficher le message "Vous avez des enfants, bravo !".



Et si la personne n'a pas d'enfants, qu'est-ce qui se passe ?

Dans le cas où le test n'est pas vérifié, l'ordinateur ne lit pas les instructions qui se trouvent entre accolades. Il saute donc à la

ligne qui suit la fermeture des accolades.

Dans notre cas, si la personne n'a aucun enfant, on verra seulement ce message apparaître :

Code : Console

```
Fin du programme
```

Faites le test ! Changez la valeur de la variable `nbEnfants`, mettez-la à 0, et regardez ce qui se passe. 😊

else : ce qu'il faut faire si la condition n'est pas vérifiée

Vous souhaitez que votre programme fasse quelque chose de précis si la condition n'est pas vérifiée ? C'est vrai que pour le moment, le programme est plutôt silencieux si vous n'avez pas d'enfants !

Heureusement, vous pouvez utiliser le mot-clé **else** qui signifie "sinon". On va par exemple afficher un autre message si la personne n'a pas d'enfants :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(0);

    if (nbEnfants > 0)
    {
        cout << "Vous avez des enfants, bravo !" << endl;
    }
    else
    {
        cout << "Eh bien alors, vous n'avez pas d'enfants ?" <<
endl;
    }

    cout << "Fin du programme" << endl;
    return 0;
}
```

Ce code affiche :

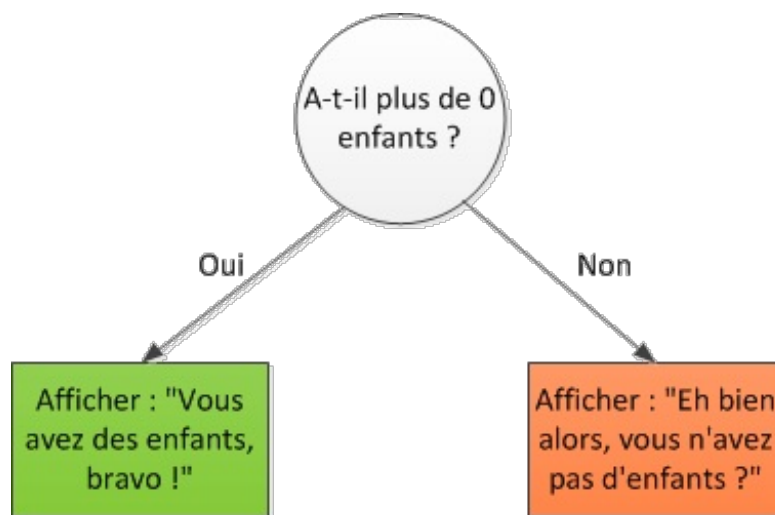
Code : Console

```
Eh bien alors, vous n'avez pas d'enfants ?
Fin du programme
```

... car j'ai changé la valeur de la variable `nbEnfants` au début, regardez bien. 😊

Si vous mettez une valeur supérieure à 0, le message redeviendra celui que nous avons vu avant !

Bien, comment ça fonctionne ? C'est très simple en fait : l'ordinateur lit d'abord la condition du **if** et se rend compte que la condition est fausse. On vérifie si la personne a au moins 1 enfant et ce n'est pas le cas. L'ordinateur "saute" tout ce qui se trouve entre les premières accolades et tombe sur la ligne du **else** qui signifie "sinon". Il effectue donc les actions indiquées après le **else**.



else if : effectuer un autre test

Il est possible de faire plusieurs tests à la suite. Imaginez qu'on souhaite faire le test suivant :

- Si le nombre d'enfants est égal à 0, afficher ce message "[...]"
- Sinon si le nombre d'enfants est égal à 1, afficher ce message "[...]"
- Sinon si le nombre d'enfants est égal à 2, afficher ce message "[...]"
- Sinon, afficher ce message "[...]"

Pour faire tous ces tests un à un dans l'ordre, on va avoir recours à la condition "else if" qui signifie "sinon si". Les tests vont être lus dans l'ordre jusqu'à ce que l'un d'entre eux soit vérifié.

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(2);

    if (nbEnfants == 0)
    {
        cout << "Eh bien alors, vous n'avez pas d'enfants ?" <<
endl;
    }
    else if (nbEnfants == 1)
    {
        cout << "Alors, c'est pour quand le deuxieme ?" << endl;
    }
    else if (nbEnfants == 2)
    {
        cout << "Quels beaux enfants vous avez la !" << endl;
    }
    else
    {
        cout << "Bon, il faut arreter de faire des gosses maintenant
!" << endl;
    }

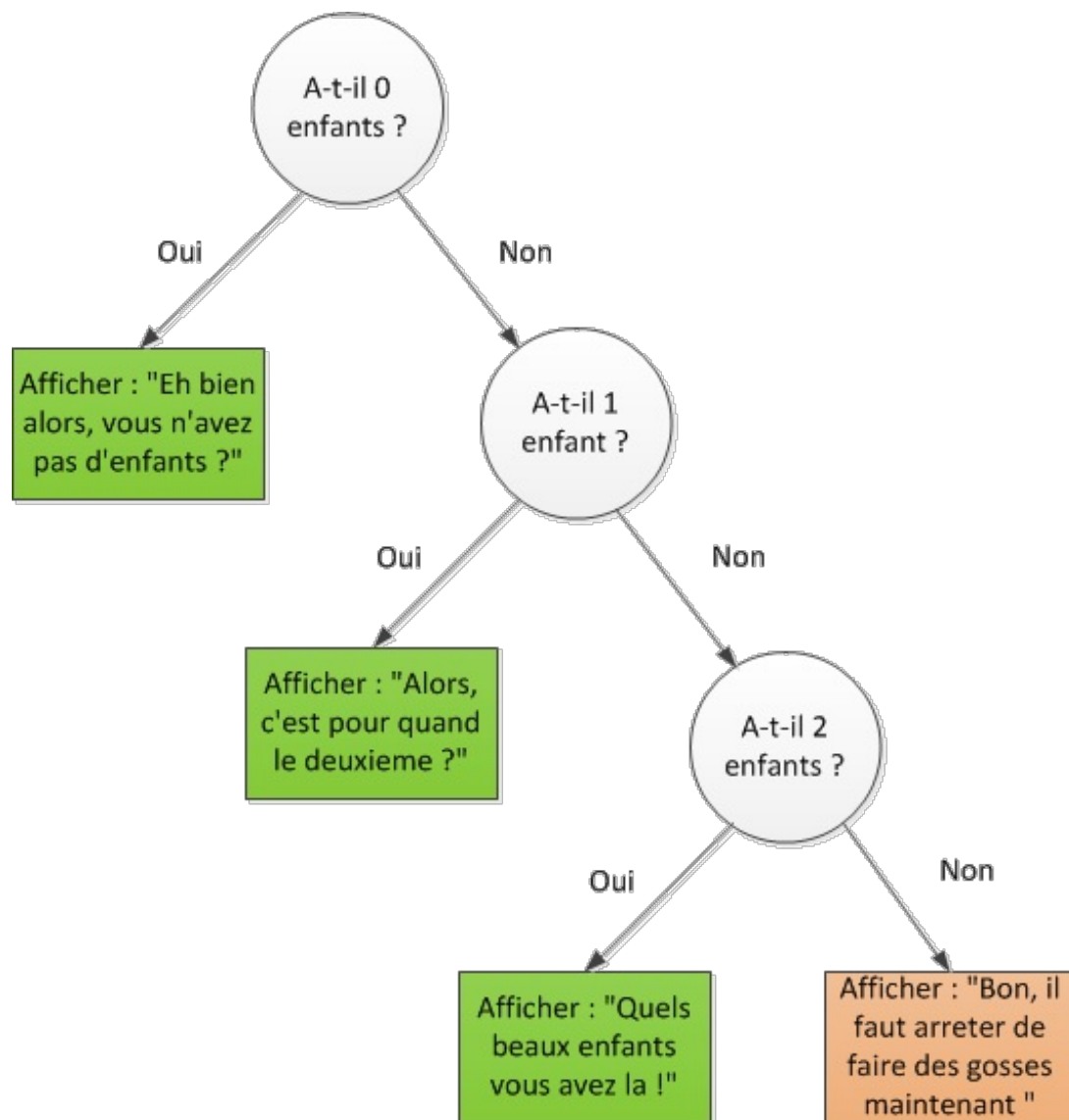
    cout << "Fin du programme" << endl;
    return 0;
}
```


Ca se complique ? Pas tant que ça. 😊

Dans notre cas, nous avons 2 enfants :

1. L'ordinateur teste d'abord si on en a 0.
2. Comme ce n'est pas le cas, il passe au premier else if : est-ce qu'on a 1 enfant ? Non plus !
3. L'ordinateur teste donc le second else if : est-ce qu'on a 2 enfants ? Oui ! Donc on affiche le message "*Quels beaux enfants vous avez la !*".

Si aucune des conditions n'avait été vérifiée, c'est le message du else "Bon, il faut arreter de faire des gosses maintenant !" qui se serait affiché.



La condition switch

En théorie, la condition if permet de faire tous les tests que l'on veut. En pratique, il existe d'autres façons de faire des tests. La condition switch, par exemple, permet de simplifier l'écriture de conditions qui testent plusieurs valeurs différentes pour une même variable.

Prenez par exemple le test qu'on vient de faire sur le nombre d'enfants :

A-t-il 0 enfants ?

A-t-il 1 enfant ?

A-t-il 2 enfants ?

...

On peut faire ce genre de tests avec des `if... else if... else`, mais on peut faire la même chose avec une condition `switch` qui a tendance à rendre le code plus lisible dans ce genre de cas. Voici ce que donnerait la condition précédente avec un `switch` :

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(2);

    switch (nbEnfants)
    {
        case 0:
            cout << "Eh bien alors, vous n'avez pas d'enfants ?" <<
endl;
            break;

        case 1:
            cout << "Alors, c'est pour quand le deuxieme ?" << endl;
            break;

        case 2:
            cout << "Quels beaux enfants vous avez la !" << endl;
            break;

        default:
            cout << "Bon, il faut arreter de faire des gosses
maintenant !" << endl;
            break;
    }

    return 0;
}
```

Cela affiche :

Code : Console

```
Quels beaux enfants vous avez la !
```

La forme est un peu différente : on indique d'abord qu'on va analyser la variable `nbEnfants`(ligne 9). Ensuite, on teste tous les cas (**case**) possibles : si ça vaut 0, si ça vaut 1, si ça vaut 2...

Les **break** sont obligatoires si on veut que l'ordinateur ne continue pas d'autres tests une fois qu'il en a vérifié un. En pratique, je vous conseille d'en mettre toujours comme moi à la fin de chaque **case**.

Enfin, le **default** à la fin correspond au **else** ("sinon") et s'exécute si aucun test précédent n'est vérifié.

Le **switch** ne permet de tester que l'égalité. Vous ne pouvez pas tester "Si le nombre d'enfants est supérieur à 2" avec un **switch** : il faut utiliser un **if** dans ce cas.

De plus, le **switch** ne peut tester que des nombres entiers (`int`, `unsigned int`, `char`). Il est impossible de tester des nombres décimaux(`double`) avec un **switch**.

Le **switch** est donc limité en terme de possibilités mais il permet d'utiliser une écriture alternative qui peut être parfois pratique dans des cas simples.



Booléens et combinaisons de conditions

Allons un peu plus loin avec les conditions. Nous allons découvrir deux notions un peu plus avancées mais néanmoins essentielles : les booléens et les combinaisons de conditions.

Les booléens

Vous vous souvenez du type `bool` ? Ce type de données peut stocker deux valeurs :

- `true` (vrai)
- `false` (faux)

Ce type est souvent utilisé avec les conditions. Quand on y pense c'est logique : une condition est soit vraie, soit fausse. Une variable booléenne aussi. 🤔

Si je vous parle du type `bool`, c'est parce qu'on peut l'utiliser d'une façon un peu particulière dans les conditions. Regardez ce code pour commencer :

Code : C++

```
bool adulte(true);

if (adulte == true)
{
    cout << "Vous etes un adulte !" << endl;
}
```

Cette condition, qui vérifie la variable `adulte`, affiche un message si celle-ci vaut vrai (`true`).

Où je veux en venir ? En fait, il est possible d'omettre la partie `"== true"` dans la condition, cela revient au même ! Regardez ce code, qui est équivalent :

Code : C++

```
bool adulte(true);

if (adulte)
{
    cout << "Vous etes un adulte !" << endl;
}
```

L'ordinateur *comprend* que vous voulez vérifier si la variable booléenne vaut `true`. Il n'est pas nécessaire de rajouter `"== true"`.



Ca ne rend pas le code plus difficile à lire ça ? 🤔

Non, au contraire le code est plus court et plus facile à lire !

`if (adulte)` se lit tout simplement "S'il est adulte".

Je vous invite à utiliser cette notation raccourcie quand vous testerez des variables booléennes. Cela vous aidera à clarifier votre code et à rendre certaines conditions plus "digestes" à lire. 🤔

Combiner des conditions

Pour les conditions les plus complexes, sachez que vous pouvez faire plusieurs tests au sein d'un seul et même **if**. Pour cela, il va falloir utiliser de nouveaux symboles :

&&	ET
	OU
!	NON

Test ET

Imaginons : on veut faire tester si la personne est adulte ET si elle a 1 enfant au moins. On va écrire :

Code : C

```
if (adulte && nbEnfants >= 1)
```

Les deux symboles "&&" signifient ET. Notre condition se dirait en français : *"Si la personne est adulte ET que le nombre d'enfants est supérieur ou égal à 1"*



Notez que j'aurais aussi pu écrire cette condition comme ceci : `if (adulte == true && nbEnfants >= 1)`. Cependant, comme je vous l'ai expliqué un peu plus tôt, il est facultatif de préciser "`== true`" sur les booléens et c'est une habitude que je vous invite à prendre.

Test OU

Pour faire un OU, on utilise les 2 signes `||`. Je dois avouer que ce signe n'est pas facilement accessible sur nos claviers. Pour le taper sur un clavier AZERTY français, il faudra faire Alt Gr + 6. Sur un clavier belge, il faudra faire Alt Gr + & et finalement, pour les Suisses, c'est la combinaison Alt Gr + 7 qu'il faut utiliser.

On peut par exemple tester si le nombre d'enfants est égal à 1 OU 2 :

Code : C

```
if (nbEnfants == 1 || nbEnfants == 2)
```

Test NON

Il faut maintenant que je vous présente un dernier symbole : le point d'exclamation. En informatique, le point d'exclamation signifie "Non".

Vous devez mettre ce signe avant votre condition pour pouvoir dire *"Si cela n'est pas vrai"* :

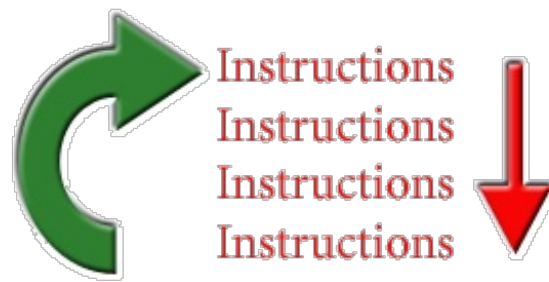
Code : C

```
if (!adulte)
```

Cela pourrait se traduire par "Si la personne n'est pas adulte".

Les boucles

Les boucles vous permettent de répéter les mêmes instructions plusieurs fois dans votre programme. Le principe est le suivant :



1. L'ordinateur lit les instructions de haut en bas (comme d'habitude)
2. Puis, une fois arrivé à la fin de la boucle, il repart à la première instruction
3. Il recommence alors à lire les instructions de haut en bas...
4. ... Et il repart au début de la boucle.

Les boucles sont répétées tant qu'une condition est vraie. Par exemple on peut faire une boucle qui dit : "Tant que l'utilisateur donne un nombre d'enfants inférieur à 0, redemander le nombre d'enfants"...

Il existe 3 types de boucles à connaître :

- while
- do ... while
- for

La boucle while

Cette boucle s'utilise comme ceci :

Code : C++

```
while (condition)
{
    /* Instructions à répéter */
}
```

Tout ce qui est entre accolades sera répété tant que la condition est vérifiée.

Essayons de faire ce que j'ai dit plus tôt : on redemande le nombre d'enfants à l'utilisateur tant que celui-ci est inférieur à 0. Ce genre de boucle permet de s'assurer que l'utilisateur rentre un nombre correct.

Code : C++

```
int main()
{
    int nbEnfants(-1); // Nombre négatif pour pouvoir rentrer dans
    la boucle

    while (nbEnfants < 0)
    {
        cout << "Combien d'enfants avez-vous ?" << endl;
        cin >> nbEnfants;
    }
}
```

```
    }

    cout << "Merci d'avoir indique un nombre d'enfants correct. Vous
en avez " << nbEnfants << endl;

    return 0;
}
```

Voici un exemple d'utilisation de ce programme :

Code : Console

```
Combien d'enfants avez-vous ?
-3
Combien d'enfants avez-vous ?
-5
Combien d'enfants avez-vous ?
2
Merci d'avoir indique un nombre d'enfants correct. Vous en avez 2
```

Tant que vous mettez un nombre négatif, la boucle recommencera. En effet, elle teste **while** (`nbEnfants < 0`) c'est-à-dire "Tant que le nombre d'enfants est inférieur à 0". Dès que le nombre devient supérieur ou égal à 0, la boucle s'arrête et le programme continue après l'accolade fermante.



Vous noterez que j'ai initialisé la variable `nbEnfants` à -1. En effet, pour que l'ordinateur veuille bien rentrer une première fois dans la boucle, il faut faire en sorte que la condition soit vraie la première fois. 😊

La boucle **do ... while**

Cette boucle est très similaire à la précédente... si ce n'est que la condition n'est analysée qu'à la fin. Cela signifie que le contenu de la boucle sera toujours lu **au moins une première fois**.

Cette boucle a cette forme :

Code : C++

```
do
{
    /* Instructions */
} while (condition);
```

Notez bien la présence du point-virgule tout à la fin !

Reprenons le code précédent et utilisons cette fois un **do ... while** :

Code : C++

```
int main()
{
    int nbEnfants(0);

    do
    {
        cout << "Combien d'enfants avez-vous ?" << endl;
        cin >> nbEnfants;
```

```
    } while (nbEnfants < 0);

    cout << "Merci d'avoir indique un nombre d'enfants correct. Vous
en avez " << nbEnfants << endl;

    return 0;
}
```

Le principe est le même, le programme a le même comportement. Le gros intérêt du do ... while est qu'on s'assure que la boucle sera lue au moins une fois.

D'ailleurs, du coup, il n'est pas nécessaire d'initialiser nbEnfants à -1 (c'est le principal avantage que procure cette solution). En effet, ici le nombre d'enfants est initialisé à 0 (comme on a l'habitude de faire), et comme la condition n'est testée qu'après le premier passage de la boucle, les instructions sont bien lues au moins une fois.

La boucle for

Ce type de boucle, que l'on retrouve fréquemment, permet de condenser :

- Une initialisation
- Une condition
- Une incrémentation

Voici sa forme :

Code : C++

```
for (initialisation ; condition ; incrementation)
{
}
}
```

Regardons un exemple concret qui affiche des nombres de 0 à 9 :

Code : C++

```
int main()
{
    int compteur(0);

    for (compteur = 0 ; compteur < 10 ; compteur++)
    {
        cout << compteur << endl;
    }

    return 0;
}
```

Ce code affiche :

Code : Console

```
0
1
2
```



```
3
4
5
6
7
8
9
```

On retrouve sur la ligne du `for` les 3 instructions que je vous ai indiquées :

- Une initialisation (`compteur = 0`) : la variable `compteur` est mise à 0 au tout début de la boucle. Notez que ça avait été fait juste la ligne au-dessus donc ce n'était pas vraiment nécessaire ici.
- Une condition (`compteur < 10`) : on vérifie que la variable `compteur` est inférieure à 10 à chaque nouveau tour de boucle.
- Une incrémentation (`compteur++`) : à chaque tour de boucle, on ajoute 1 à la variable `compteur` ! Voilà pourquoi on voit s'afficher à l'écran des nombres de 0 à 9. 😊



Vous pouvez faire autre chose qu'une incrémentation si vous le désirez. La dernière section du **for** est réservée à la modification de la variable et vous pouvez donc y faire une décrémentation (`compteur--`) ou avancer de 2 en 2 (`compteur += 2`), etc.

Notez qu'il est courant de déclarer la variable directement à l'intérieur du `for`, comme ceci :

Code : C++

```
int main()
{
    for (int compteur(0) ; compteur < 10 ; compteur++)
    {
        cout << compteur << endl;
    }

    return 0;
}
```

La variable n'existe alors que pendant la durée de la boucle `for`. C'est la forme la plus courante de la boucle **for**. On ne déclare la variable avant le **for** que si on en a besoin plus tard. C'est un cas assez rare.



Quand utiliser un `for` et quand utiliser un `while` ?

On utilise la boucle `for` quand on connaît le nombre de fois que l'on souhaite boucler, et on utilise le plus souvent la boucle `while` quand on ne sait pas combien de fois la boucle va être effectuée.

Vous savez maintenant manier les structures de contrôle, ces éléments indispensables à tous les programmes informatiques ! 😊

Prenez le temps de vous approprier les codes présentés dans ce chapitre car nous allons utiliser des conditions et des boucles tout au long de ce cours !

Découper son programme en fonctions

Nous venons de voir comment faire varier le déroulement d'un programme en utilisant des boucles et des branchements. Avant ça, je vous ai parlé des variables. Ce sont des éléments qui se retrouvent dans tous les langages de programmation. C'est aussi le cas de la notion que nous allons aborder dans ce chapitre : les **fonctions**.

Tous les programmes C++ utilisent des fonctions et vous en avez aussi utilisé plusieurs sans le savoir. 😊

Le but des fonctions est de découper son programme en petits éléments réutilisables, un peu comme des briques. On peut les assembler d'une certaine manière pour créer un mur, ou d'une autre manière pour faire un cabanon ou même un gratte-ciel. Une fois que les briques sont créées, la tâche du programmeur ne consiste "plus qu'à" les assembler.

Commençons par créer des briques. Nous apprendrons à les utiliser dans un deuxième temps.

Créer et utiliser une fonction

Dès le début de ce cours, nous avons utilisé des fonctions, toujours la même en fait. La fonction `main()`. C'est le point d'entrée de tous les programmes C++, c'est par là que tout commence.

Code : C++

```
#include <iostream>
using namespace std;

int main() //Debut de la fonction main() et donc du programme
{
    cout << "Bonjour tout le monde !" << endl;
    return 0;
} //Fin de la fonction main() et donc du programme
```

Le programme commence réellement à la ligne 4 et se termine à la ligne 8 après l'accolade fermante. C'est-à-dire que tout se déroule dans une seule et unique fonction. On n'en sort pas. Il n'y a qu'une seule portion de code qui est exécutée dans l'ordre sans jamais sauter ailleurs.

Alors, si je vous dis tout ça, c'est vous vous en doutez, que l'on peut écrire d'autres fonctions. Et donc avoir un programme découpé en plusieurs modules indépendants.



Pourquoi vouloir faire ça ?

C'est vrai après tout, mettre l'entièreté du code dans la fonction `main()` est tout à fait possible. Ce n'est cependant pas une bonne pratique.

Imaginons que nous voulions créer un jeu vidéo 3D. Comme c'est quand même assez complexe, le code source va nécessiter plusieurs dizaines de milliers de lignes ! 😞 Si l'on garde tout dans une seule fonction, il va être très difficile de s'y retrouver. Il serait certainement plus simple d'avoir un morceau de code dans un coin qui fait bouger un personnage et un autre bout de code ailleurs qui charge les niveaux, etc. Découper son programme en fonctions permet de s'**organiser**.

En plus, si vous êtes plusieurs programmeurs à travailler sur le même programme, vous pourrez vous partager plus facilement le travail. Chacun travaille sur une fonction différente.

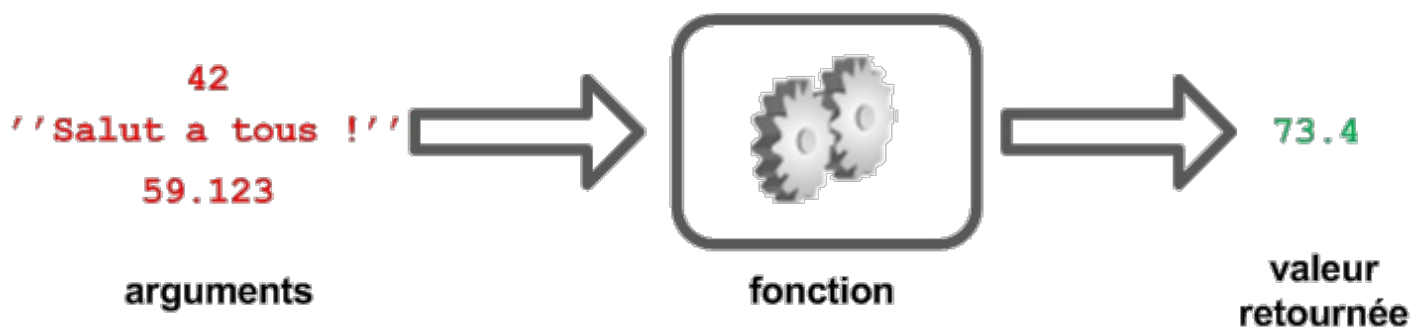
Mais ce n'est pas tout !

Prenons par exemple le calcul de la racine carrée. Si vous créez un programme de maths, il est bien possible que vous ayez besoin à plusieurs endroits d'effectuer des calculs de racine. Avoir une fonction `sqrt()` va nous permettre de faire plusieurs de ces calculs sans avoir à recopier le même code à plusieurs endroits. On peut **utiliser plusieurs fois la même fonction** et c'est une des raisons principales d'en écrire.

Présentation des fonctions

Une fonction est un morceau de code qui accomplit une tâche particulière. Elle reçoit des données à traiter, effectue des actions avec et finalement renvoie une valeur.

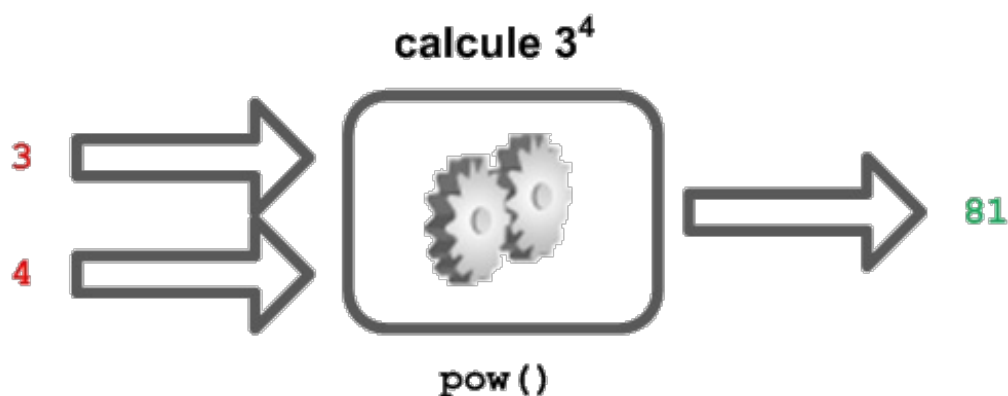
Les données entrantes s'appellent les **arguments** et on utilise l'expression **valeur retournée** pour les éléments qui sortent de la fonction.



Vous vous souvenez de `pow()` ? La fonction qui permet de calculer des puissances ? En utilisant le nouveau vocabulaire, on peut dire que cette fonction :

1. reçoit deux arguments.
2. effectue un calcul mathématique.
3. renvoie le résultat du calcul.

En utilisant un schéma comme le précédent, on peut imaginer `pow()` comme ceci :



Vous en avez déjà fait l'expérience, on peut utiliser cette fonction plusieurs fois. Ce qui implique que nous ne sommes pas obligés de copier le code (compliqué) qui se trouve à l'intérieur de `pow()` à chaque fois que l'on souhaite effectuer un calcul de puissance.

Définir une fonction

Bon bon, il est temps d'attaquer le concret. Il faut que je vous montre comment définir une fonction. Je pourrais vous dire de regarder comment `main()` est fait et vous laisser patauger, mais je suis sympa. Je vais vous guider 😊

Vous êtes prêt ? Alors allons-y !

Toutes les fonctions ont la forme suivante :

Code : C++

```
type nomFonction(arguments)
{
    //Instructions effectuées par la fonction
}
```

On retrouve les trois éléments dont je vous ai déjà parlé auxquels s'ajoute le nom de la fonction.

- Le premier élément est le **type de retour**. Il permet d'indiquer le type de variable renvoyé par la fonction. Si votre fonction doit renvoyer du texte, alors ce sera `string`, si votre fonction effectue un calcul, alors ce sera `int` ou `double`.
- Le deuxième élément est le **nom de la fonction**. Vous connaissez déjà `main()`, `pow()` ou `sqrt()`. L'important est de choisir un nom de fonction qui décrit bien ce que fait la fonction. Comme pour les variables en fait. 😊
- Entre les parenthèses, on trouve la **liste des arguments** de la fonction. Ce sont les données avec lesquelles la fonction va travailler. Il peut y avoir un argument (comme pour `sqrt()`), plusieurs arguments (comme pour `pow()`) ou aucun argument (comme pour `main()`).
- Finalement, il y a des **accolades** qui délimitent le contenu de la fonction. Toutes les opérations qui seront effectuées se trouvent entre les deux accolades.

Il est possible de créer **plusieurs fonctions ayant le même nom**. Il faut alors que la liste des arguments des deux fonctions soit différente. C'est ce qu'on appelle la **surcharge** d'une fonction.



Dans un même programme, il peut par exemple y avoir la fonction `int addition(int a, int b)` et la fonction `double addition(double a, double b)`. Les deux fonctions ont le même nom mais une travaille avec des entiers et l'autre avec des nombres réels.

Créons donc des fonctions !

Une fonction toute simple

Commençons par une fonction basique. Une fonction qui reçoit un nombre entier, ajoute 2 à ce nombre et le renvoie.

Code : C++

```
int ajouteDeux(int nombreRecu)
{
    int valeur(nombreRecu + 2); //On cree une case en memoire.
                                //On prend le nombre reçu en
argument, on y ajoute 2.
                                //Et on met tout ça dans la memoire.

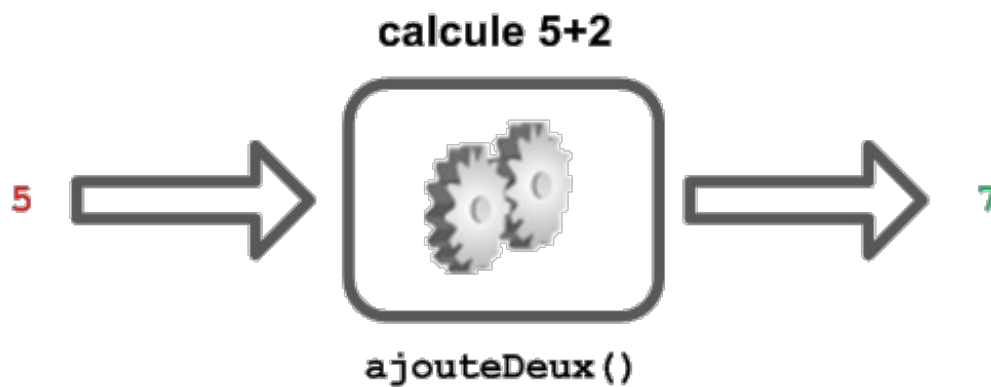
    return valeur;              //On indique que la valeur qui sort de
la fonction                    //est la valeur de la variable
    'valeur'
}
```



Il n'y a pas de point-virgule ! Ni après la déclaration, ni après l'accolade fermante.

Analysons ce code en détail. Il y a deux lignes vraiment nouvelles pour vous.

Avec ce que je vous ai expliqué, vous devriez comprendre la première ligne. On déclare une fonction nommée `ajouteDeux` qui va recevoir un nombre entier en argument et qui, une fois qu'elle aura terminé, va recracher un autre nombre entier.



Toutes les lignes suivantes utilisent des choses déjà connues sauf l'avant-dernière. Si vous vous posez des questions sur ces lignes, je vous invite à relire le chapitre sur [l'utilisation de la mémoire](#).

Le **return** de l'avant-dernière ligne indique quelle valeur va ressortir de la fonction. En l'occurrence, c'est la valeur de la variable `valeur` qui va être renvoyée.

Appeler une fonction

Bon, c'est bien joli tout ça, mais il faut encore apprendre à l'utiliser cette fonction. C'est vrai, mais vous savez déjà le faire. Souvenez-vous des fonctions mathématiques !

Code : C++

```
#include <iostream>
using namespace std;

int ajouteDeux(int nombreRecu)
{
    int valeur(nombreRecu + 2);

    return valeur;
}

int main()
{
    int a(2), b(2);
    cout << "Valeur de a : " << a << endl;
    cout << "Valeur de b : " << b << endl;
    b = ajouteDeux(a); //Appel de la fonction
    cout << "Valeur de a : " << a << endl;
    cout << "Valeur de b : " << b << endl;

    return 0;
}
```

On retrouve la syntaxe que l'on connaissait déjà : `résultat = fonction(argument)`. Facile pour ainsi dire ! 😊
Vous avez essayé le programme ? Vous devriez toujours essayer les exemples. Voici quand même ce que ça donne :

Code : Console

```
Valeur de a : 2
Valeur de b : 2
Valeur de a : 2
Valeur de b : 4
```

Après l'appel à la fonction, la variable `b` a été modifiée. Tout fonctionne donc comme annoncé.

Plusieurs paramètres

On n'est pas encore au bout de nos peines. Il y a des fonctions qui prennent plusieurs paramètres, comme `pow()` et `getline()` par exemple.

Pour passer plusieurs paramètres à une fonction, il faut les séparer par des virgules.

Code : C++

```
int addition(int a, int b)
{
    return a+b;
}

double multiplication(double a, double b, double c)
{
    return a*b*c;
}
```

La première de ces fonctions calcule la somme des deux nombres qui lui sont fournis alors que la deuxième calcule le produit des trois nombres reçus.



Vous pouvez bien sûr écrire des fonctions qui prennent des arguments de type différent. 🤖

Pas d'arguments

A l'inverse, il est aussi possible de créer des fonctions sans arguments. 🤖 Il faut simplement ne rien écrire entre les parenthèses !



Mais à quoi ça sert ?

On peut imaginer plusieurs scénarios, mais pensez par exemple à une fonction qui demande à l'utilisateur d'entrer son nom. Elle n'a pas besoin de paramètres.

Code : C++

```
string demanderNom()
{
    cout << "Entrez votre nom : ";
    string nom;
    cin >> nom;
    return nom;
}
```

Je suis sûr que vous trouverez plein d'exemples par la suite ! Même si c'est vrai que ce type de fonctions est plus rare. 🤖

Des fonctions qui ne renvoient rien

Tous les exemples que je vous ai donnés jusque-là prenaient des arguments et renvoyaient une valeur. Mais il est aussi possible d'écrire des fonctions qui ne renvoient rien. Enfin presque.

Rien ne ressort de la fonction, mais quand on la déclare, il faut quand même indiquer un type. On utilise le "type" `void`, ce qui signifie *néant* en anglais. Ça veut tout dire, il n'y a réellement rien qui ressort de la fonction. 🤔

Code : C++ - Une fonction ne renvoyant rien

```
void direBonjour()
{
    cout << "Bonjour !" << endl;
    //Comme rien ne ressort, il n'y a pas de return !
}

int main()
{
    direBonjour();    //Comme la fonction ne renvoie rien, on
    l'appelle          //sans mettre la valeur de retour dans une
    variable
    return 0;
}
```

Avec ce dernier point, nous avons fait le tour de la théorie. Dans la suite du chapitre, je vous propose quelques exemples et un super schéma récapitulatif. Ce n'est pas le moment de partir.

Quelques exemples

Le carré

Commençons de manière simple. Calculer le carré d'un nombre. Cette fonction reçoit un nombre x en argument et calcule la valeur de x^2 .

Code : C++

```
#include <iostream>
using namespace std;

double carre(double x)
{
    double resultat;
    resultat = x*x;
    return resultat;
}

int main()
{
    double nombre, carreNombre;
    cout << "Entrez un nombre : ";
    cin >> nombre;

    carreNombre = carre(nombre); //On utilise la fonction

    cout << "Le carré de " << nombre << " est " << carreNombre <<
    endl;
    return 0;
}
```

Je vous avais promis un schéma, le voilà. Voyons ce qui se passe dans ce programme et dans quel ordre les lignes sont exécutées.

1) Le programme commence au début de la fonction `main()`.

2) Le programme exécute les 3 premières lignes comme d'habitude.

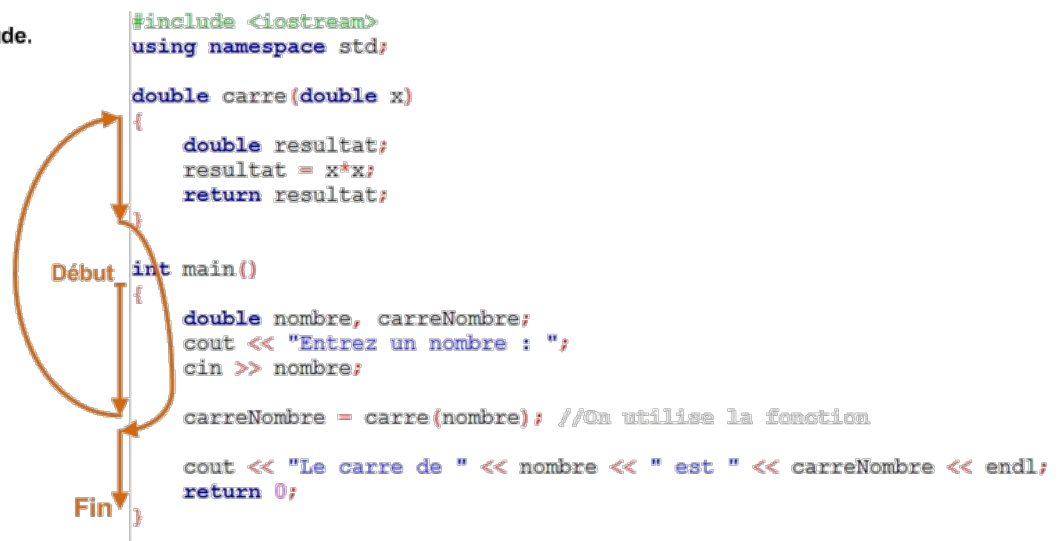
3) Le programme repère un appel de fonction.

4) Il évalue la valeur de l'argument. Cette valeur est la valeur de nombre. Elle est recopiée dans la case mémoire `x`.

5) Le programme saute au début de la fonction `carre()`. Il exécute le code de celle-ci comme d'habitude.

6) Le programme arrive à la fin de `carre()`. Il copie la valeur de résultat dans la case mémoire `carreNombre`.

7) Le programme retourne dans la fonction `main()` et exécute les dernières lignes.



Il y a une chose dont il faut absolument se rappeler. Les valeurs des variables transmises aux fonctions sont **copiées dans de nouvelles cases mémoires**. La fonction `carre()` n'agit donc pas sur les variables déclarées dans la fonction `main()`. Elle travaille uniquement avec ses propres cases mémoires.

Ce n'est que lors du **return** que les variables de `main()` sont modifiées. Ici la variable `carreNombre`. La variable `nombre` reste **inchangée** lors de l'appel à la fonction.

Réutiliser la même fonction

L'intérêt d'utiliser une fonction ici est bien sûr de pouvoir calculer facilement le carré de différents nombres. Par exemple de tous les nombres entre 1 et 20 :

Code : C++ - Calcul des carrés des nombres entre 1 et 20

```

#include <iostream>
using namespace std;

double carre(double x)
{
    double resultat;
    resultat = x*x;
    return resultat;
}

int main()
{
    for(int i(1); i<=20 ; i++)
    {
        cout << "Le carre de " << i << " est : " << carre(i) <<
    endl;
    }
    return 0;
}

```

On écrit une seule fois la "formule" du calcul du carré et ensuite on utilise cette "brique" vingt fois. Ici, le calcul est simple, mais dans bien des cas, utiliser une fonction raccourcit beaucoup le code !

Une fonction à deux arguments

Avant de terminer cette partie, voici un dernier exemple. Cette fois, je vous propose une fonction utilisant deux arguments. Nous allons dessiner un rectangle d'étoiles * dans la console. La fonction a besoin de deux arguments : la largeur et la hauteur du rectangle.

Code : C++ - Dessin d'un rectangle d'étoiles

```
#include <iostream>
using namespace std;

void dessineRectangle(int l, int h)
{
    for(int ligne(0); ligne < h; ligne++)
    {
        for(int colonne(0); colonne < l; colonne++)
        {
            cout << "*";
        }
        cout << endl;
    }
}

int main()
{
    int largeur, hauteur;
    cout << "Largeur du rectangle : ";
    cin >> largeur;
    cout << "Hauteur du rectangle : ";
    cin >> hauteur;

    dessineRectangle(largeur, hauteur);
    return 0;
}
```

Une fois compilé ce programme s'exécute et donnera par exemple :

Code : Console

```
Largeur du rectangle : 16
Hauteur du rectangle : 3
*****
*****
*****
```

Voilà la première version d'un logiciel de dessin révolutionnaire !

Cette fonction ne fait qu'afficher du texte. Elle n'a donc pas besoin de renvoyer quelque chose. C'est pour ça, qu'elle est déclarée avec le "type" `void`.

On peut facilement modifier la fonction pour qu'elle renvoie la surface du rectangle. A ce moment-là, il faudra qu'elle renvoie un `int`.

Essayez de modifier cette fonction ! 🐱 Voici deux idées :

- Afficher un message d'erreur si la hauteur ou la largeur est négative.
- Ajouter un argument pour le symbole à utiliser lors du dessin.

Amusez-vous bien. C'est important de bien maîtriser tous ces concepts.

La fin de ce chapitre est consacrée à trois notions un peu plus avancées. Vous pourrez toujours y revenir plus tard si nécessaire. Mais n'oubliez pas le QCM ! 🤖

Passage par valeur et passage par référence

La première des choses avancées dont il faut que je vous parle c'est la manière dont l'ordinateur gère la mémoire avec les fonctions.

Passage par valeur

Prenons une fonction simple qui ajoute simplement deux à son argument. Vous commencez à bien la connaître. Je l'ai donc modifiée un poil. 🤖

Code : C++

```
int ajouteDeux(int a)
{
    a+=2;
    return a;
}
```



Utiliser += ici est volontairement bizarre ! Vous verrez tout de suite pourquoi.

Testons donc cette fonction. Je pense ne rien vous apprendre en vous proposant le code suivant.

Code : C++

```
#include <iostream>
using namespace std;

int ajouteDeux(int a)
{
    a+=2;
    return a;
}

int main()
{
    int nombre(4), resultat;
    resultat = ajouteDeux(nombre);

    cout << "Le nombre original vaut : " << nombre << endl;
    cout << "Le resultat vaut : " << resultat << endl;
    return 0;
}
```

Ce qui donne sans surprise :

Code : Console

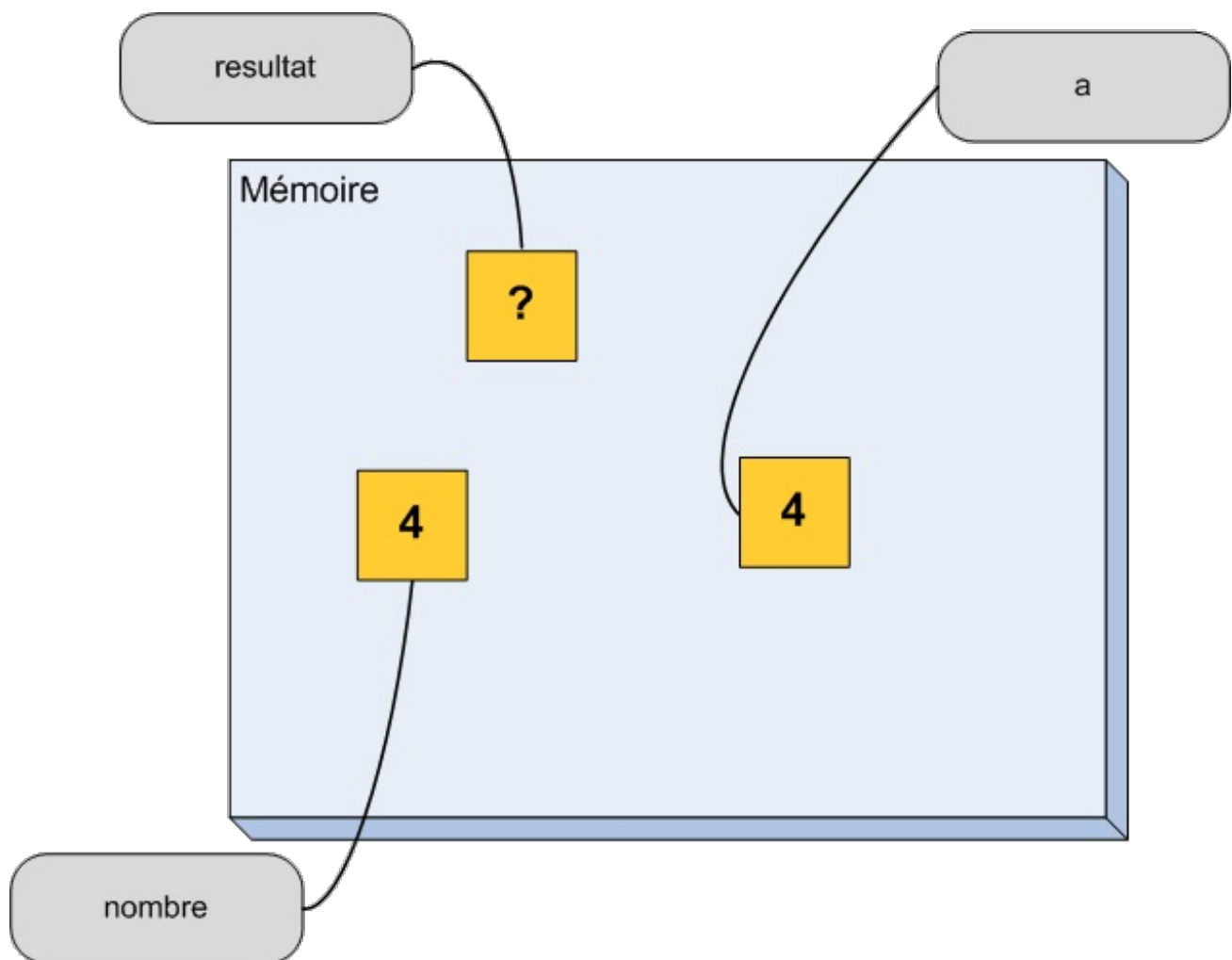
```
Le nombre original vaut : 4
Le resultat vaut : 6
```

L'étape intéressante est bien sûr ce qui se passe à la ligne 13. Vous vous rappelez des schémas de la mémoire ? Il est temps de les ressortir.

Lors de l'appel à la fonction, il se passe énormément de choses :

1. Le programme évalue la valeur de `nombre`. Il trouve **4**.
2. Le programme **alloue un nouvel espace** dans la mémoire et y écrit la valeur **4**. Cet espace mémoire possède l'étiquette `a`, le nom de la variable dans la fonction.
3. Le programme entre dans la fonction.
4. Le programme ajoute **2** à la variable `a`.
5. La valeur de `a` est ensuite copiée et assignée à la variable `resultat`, qui vaut donc maintenant **6**.
6. On sort alors de la fonction.

Ce qui est important, c'est que la variable `nombre` est copiée dans une nouvelle case mémoire. On dit que `a` est **passé par valeur**. Lorsque le programme se situe dans la fonction, la mémoire ressemble donc à ce qui se trouve sur ce schéma :



On se retrouve donc avec trois cases dans la mémoire. L'autre élément important est que la variable `nombre` va rester inchangée.

Si j'insiste sur ces points, c'est bien sûr que l'on peut faire autrement. 🤔

Passage par référence

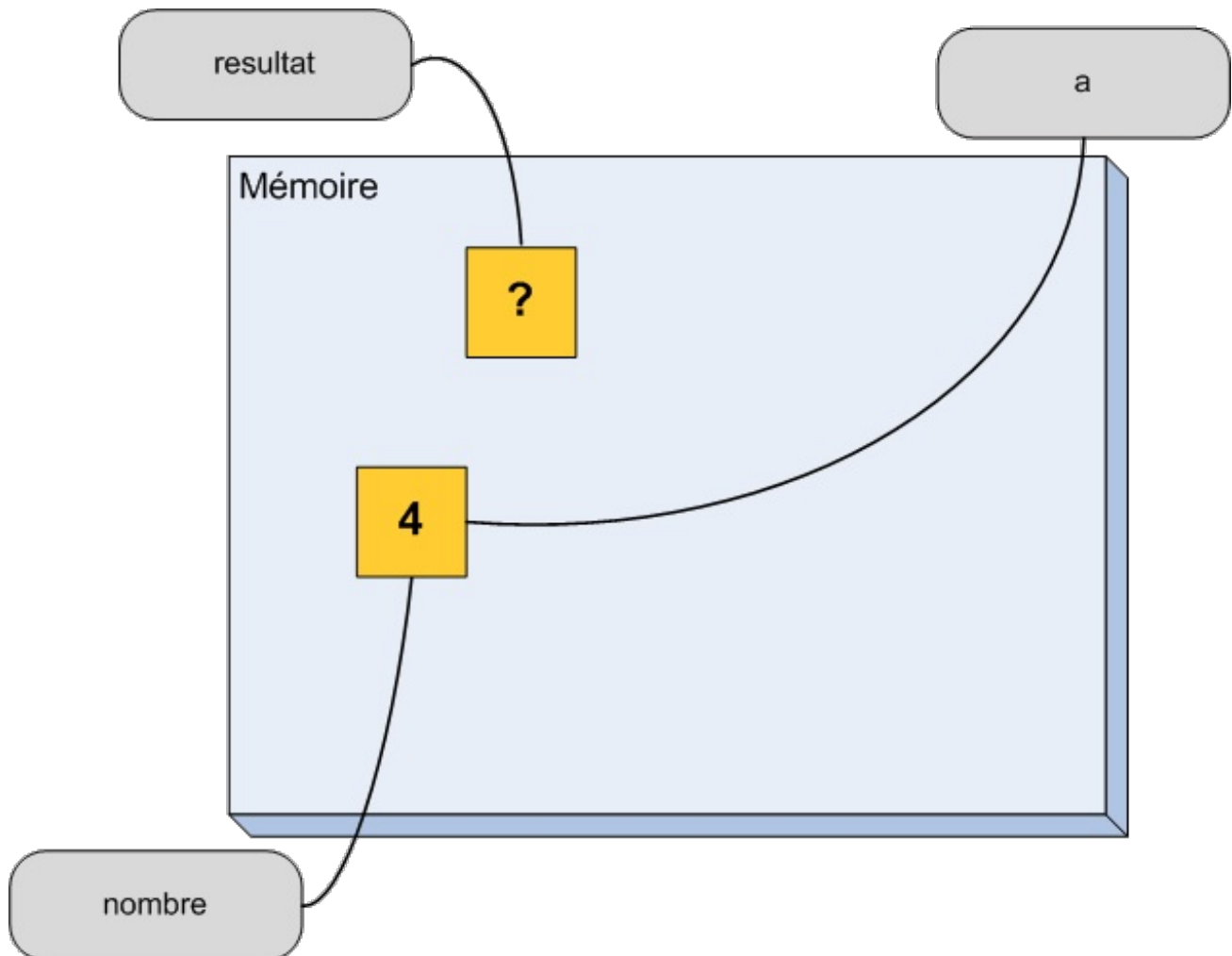
Vous vous rappelez des références ? 🧐 Oui, oui, ces choses bizarres dont je vous ai parlé il y a quelques chapitres. Si vous n'êtes pas sûr de vous, n'hésitez-pas à vous [rafraichir la mémoire](#). C'est le moment de voir à quoi servent ces drôles de bêtes.

Plutôt que de copier la valeur de `nombre` dans la variable `a`, il est possible d'ajouter une "deuxième étiquette" à la variable `nombre` à l'intérieur de la fonction. Et c'est bien sûr une référence qu'il faut utiliser comme argument de la fonction.

Code : C++

```
int ajouteDeux(int& a) //Notez le petit & !!
{
    a+=2;
    return a;
}
```

Lorsque l'on appelle la fonction, il n'y a plus de copie. Le programme donne juste un alias à la variable `nombre`. Jetons un œil à la mémoire dans ce cas.



Cette fois, la variable `a` et la variable `nombre` sont confondues. On dit que l'argument `a` est **passé par référence**.



Quel intérêt y a-t-il à faire un passage par référence ?

Cela va permettre à la fonction `ajouteDeux()` de modifier ses arguments ! Elle va ainsi pouvoir avoir une influence durable sur le reste du programme. Essayons pour voir. Reprenons le programme précédent, mais avec une référence comme argument. On obtient cette fois :

Code : Console

```
Le nombre original vaut : 6
Le resultat vaut : 6
```

Que s'est-il passé ? C'est à la fois simple et compliqué. 😞

Comme `a` et la variable `nombre` correspondent à la même case mémoire, faire `a+=2` a modifié la valeur de `nombre` ! Utiliser des références peut donc être très dangereux. C'est pour cela qu'on ne les utilise que quand on en a réellement besoin.



Justement, est-ce qu'on pourrait avoir un exemple utile ?

J'y viens, j'y viens. Ne soyez pas trop pressés. 😊

L'exemple classique est la fonction `echange()`. C'est une fonction qui échange les valeurs des deux arguments qu'on lui fournit :

Code : C++

```
void echange(double& a, double& b)
{
    double temporaire(a);           //On sauvegarde la valeur de 'a'
    a = b;                           //On remplace la valeur de 'a' par
    celle de 'b'                     //Et on utilise la valeur
    b = temporaire;                  sauvegardée pour mettre l'ancienne valeur de 'a' dans 'b'
}

int main()
{
    double a(1.2), b(4.5);

    cout << "a vaut " << a << " et b vaut " << b << endl;

    echange(a,b);    //On utilise la fonction

    cout << "a vaut " << a << " et b vaut " << b << endl;
    return 0;
}
```

Ce code fournit le résultat suivant :

Code : Console

```
a vaut 1.2 et b vaut 4.5
a vaut 4.5 et b vaut 1.2
```

Les valeurs des deux variables ont été échangées.

Si l'on n'utilisait pas un passage par référence, alors ce seraient des *copies* des arguments qui seraient échangées. Et pas les vrais arguments. Cette fonction serait donc inutile.

Je vous invite à tester cette fonction avec et sans les références. Vous verrez ainsi précisément ce qui se passe.

A priori, le passage par référence peut vous sembler obscur et compliqué. Vous verrez par la suite qu'il est souvent utilisé. Vous pourrez toujours revenir lire cette partie plus tard si ce n'est pas vraiment clair dans votre esprit.

Je vous rassure, il m'a fallu un moment pour vraiment saisir tout ça. 😊

Avancé: Le passage par référence constante

Puisque nous parlons de références, il faut quand même que je vous présente une application bien pratique. En fait, cela nous sera surtout utile dans la suite de ce cours, mais nous pouvons déjà prendre un peu d'avance.

Le passage par référence offre un gros avantage sur le passage par valeur, aucune copie n'est effectuée. Imaginez une fonction recevant en argument une `string`. Si votre chaîne de caractère contient un très long texte (l'entièreté de ce cours par exemple !), alors la copie va prendre du temps, même si tout cela se passe uniquement dans la mémoire de l'ordinateur. Cette copie est totalement inutile et il serait donc bien de pouvoir l'éliminer pour améliorer les performances du programme. Et c'est là que vous me dites : "Utilisons un passage par référence !". Oui, c'est une bonne idée. En utilisant un passage par référence, aucune copie n'est effectuée. Seulement, cette manière de procéder a un petit défaut : elle autorise la modification de l'argument. Et oui, c'est justement dans ce but que les références existent.

Code : C++

```
void f1(string texte) //Implique une copie coûteuse de 'texte'
{
}

void f2(string& texte) //Implique que la fonction peut modifier
'texte'
{
}
```

La solution est d'utiliser ce que l'on appelle un **passage par référence constante**. On évite la copie en utilisant une référence et l'on empêche la modification de l'argument en le déclarant constant. 😎

Code : C++

```
void f1(string const& texte) //Pas de copie et pas de modification
possible
{
}
```

Pour l'instant, cela peut vous sembler obscur et plutôt inutile. Dans la partie II de ce cours, nous aborderons la POO et nous utiliserons très souvent cette technique. Vous pourrez toujours revenir lire ces lignes à ce moment-là. 😊

Utiliser plusieurs fichiers

Dans l'introduction, je vous ai dit que le but des fonctions était de pouvoir réutiliser les briques créées dans plusieurs programmes.

Pour le moment, les fonctions que vous savez créer se situent à côté de la fonction `main()`. On ne peut donc pas vraiment les réutiliser.

Le C++ permet de découper son programme en plusieurs fichiers sources. Chaque fichier contient une ou plusieurs fonctions. On peut alors inclure les fichiers, et donc les fonctions, dont on a besoin dans différents projets. On a ainsi réellement des briques séparées utilisables pour construire différents programmes.

Les fichiers nécessaires

Pour faire les choses proprement, il ne faut pas un, mais deux fichiers 😊 :

- Un fichier source dont l'extension est `.cpp`. Ce fichier contient le code source de la fonction.
- Un fichier d'en-tête dont l'extension est `.h`. Ce deuxième fichier contient uniquement la description de la fonction. Ce qu'on appelle le **prototype** de la fonction.

Créons donc ces deux fichiers pour notre célèbre fonction `ajouteDeux()` :

Code : C++

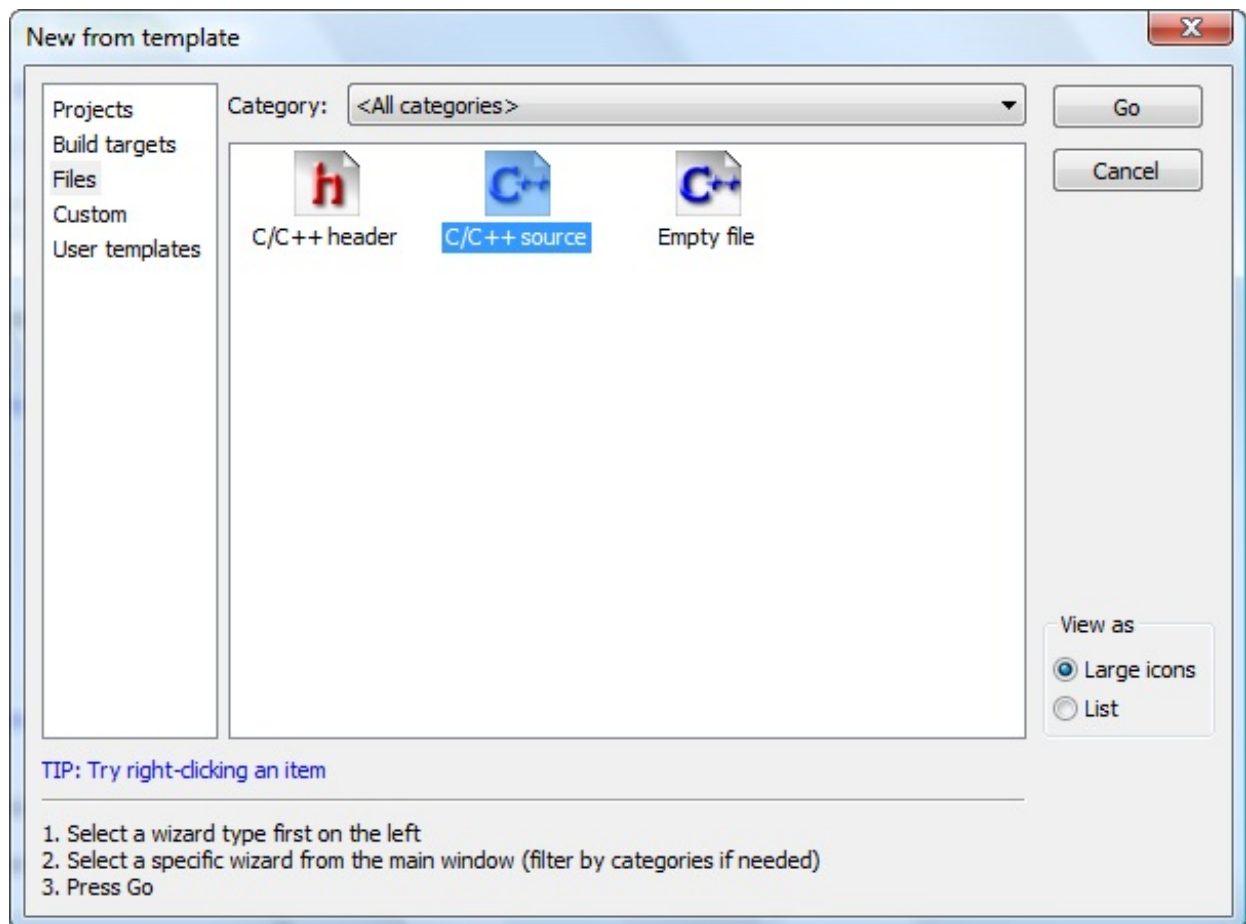
```
int ajouteDeux(int nombreRecu)
```



```
{  
    int valeur(nombreRecu + 2);  
    return valeur;  
}
```

Le fichier source

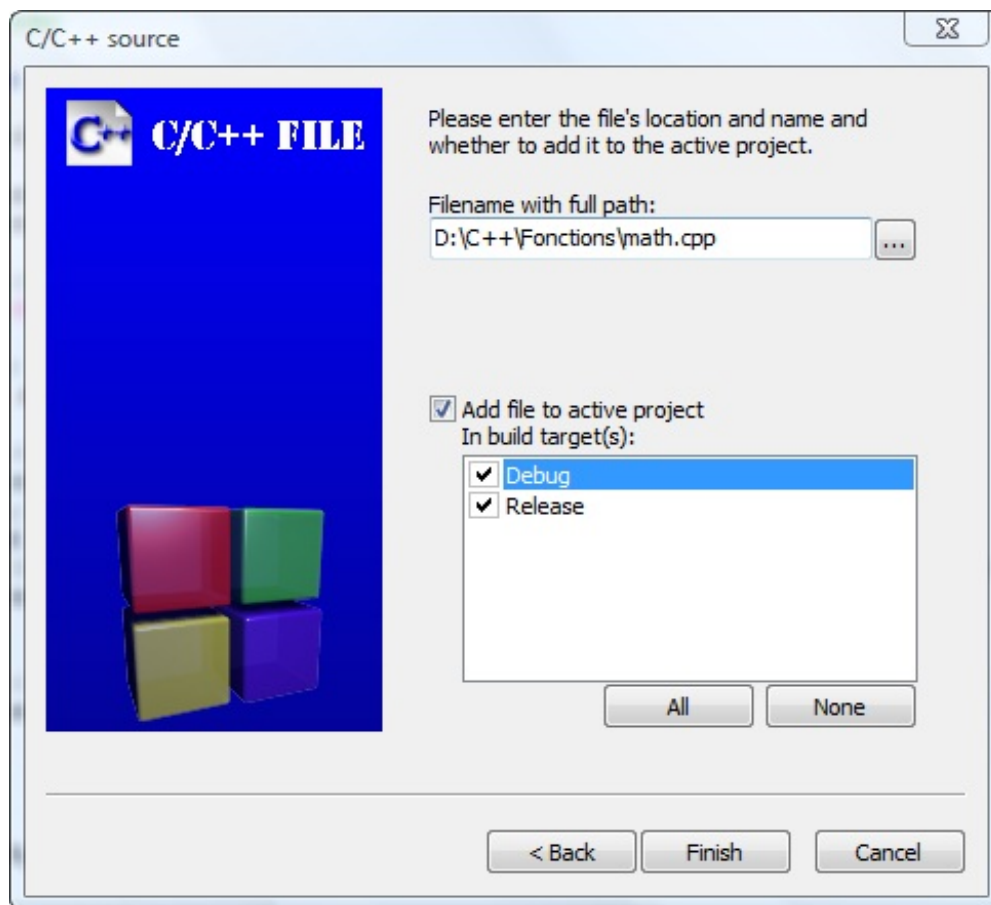
C'est le plus simple des deux. Allez dans le menu File / New / File. Choisissez ensuite C/C++ source.



Cliquez ensuite sur "Go". Comme lors de la création du projet, le programme vous demande ensuite si vous faites du C ou du C++. Choisissez "C++" bien sûr ! 😊

Finalement, on vous demande le nom du fichier à créer. Comme pour tout, il vaut mieux choisir un nom intelligent pour ses fichiers. On peut ainsi mieux s'y retrouver. Pour la fonction `ajouteDeux()`, je choisis le nom `math.cpp` et je place le fichier dans le même dossier que mon fichier `main.cpp`.

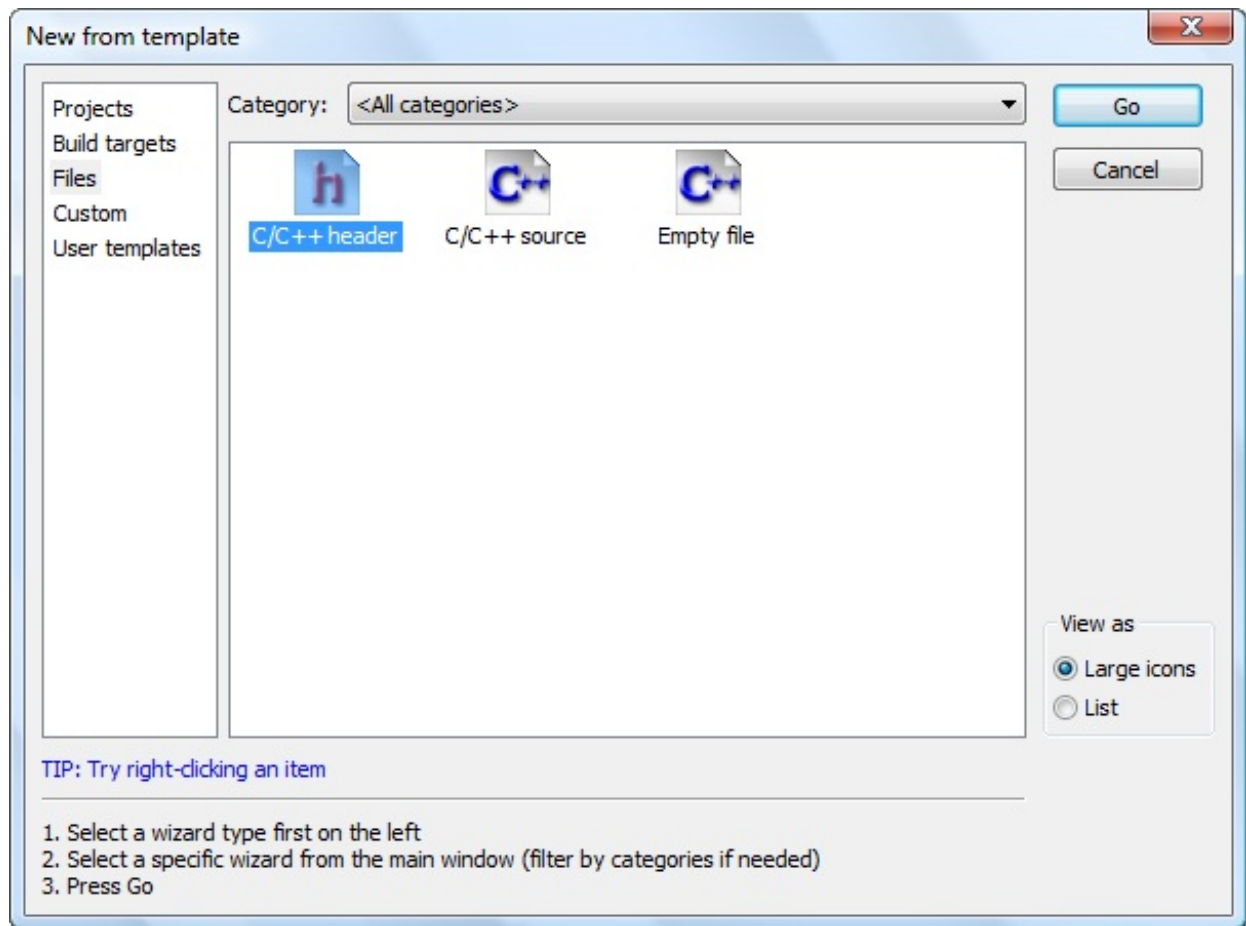
Cochez ensuite toutes les options.



Cliquez sur "Finish". Votre fichier source est maintenant créé. Passons au fichier d'en-tête.

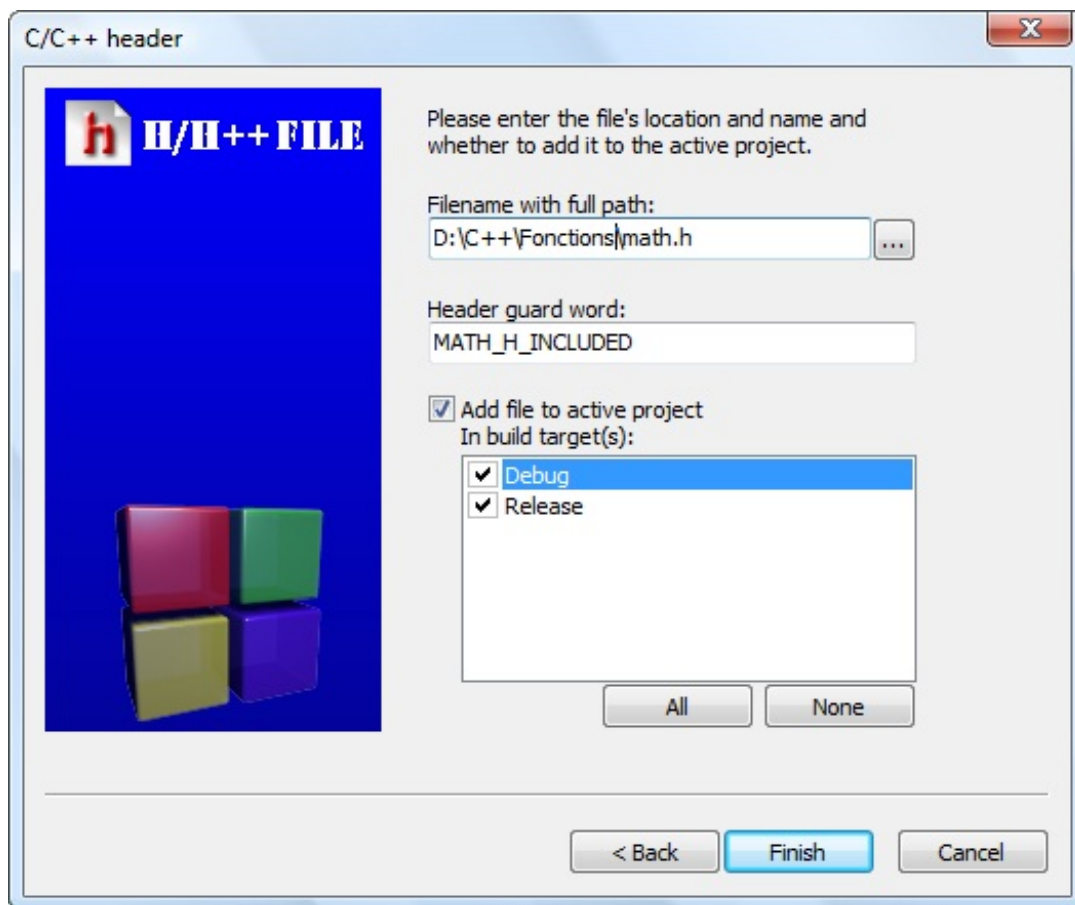
Le fichier d'en-tête

Le début est quasiment identique. Ouvrez le menu File / new / File. Sélectionnez ensuite "C/C++ header".



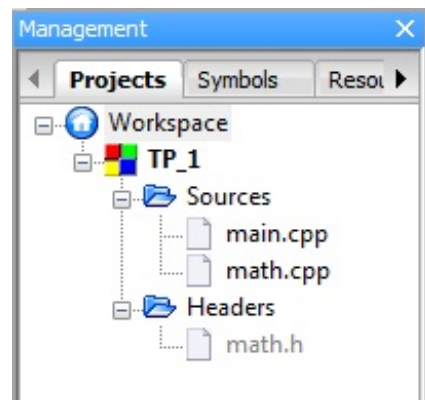
Dans la fenêtre suivante, indiquez le nom du fichier à créer. Il est conseillé d'utiliser le **même nom** que pour le fichier source mais avec une extension *.h* au lieu de *.cpp*. Dans notre cas, ce sera *math.h*. Placez le fichier dans le même dossier que les deux autres.

Ne touchez pas le champ juste en-dessous et n'oubliez pas de cocher toutes les options.



Cliquez sur "Finish". Et voilà. 😊

Une fois que les deux fichiers sont créés, vous devriez les voir apparaître dans la colonne de gauche de Code::Blocks :



Le nom du projet sera certainement différent dans votre cas. 😊

Déclarer la fonction dans les fichiers

Maintenant que nous avons nos fichiers, il ne reste qu'à les remplir.

Le fichier source

Je vous ai dit que le fichier source contenait la déclaration de la fonction. C'est un des éléments.

L'autre est plus compliqué à comprendre. Le compilateur a besoin de savoir que les fichiers `.cpp` et `.h` ont un lien entre eux. Il faut

donc commencer le fichier par la ligne suivante :

Code : C++

```
#include "math.h"
```

Vous reconnaissez certainement cette ligne. Elle indique que l'on va utiliser ce qui se trouve dans le fichier *math.h*.



Il faut utiliser des guillemets " ici et pas des chevrons < et > comme vous en aviez l'habitude jusque là.

Le fichier *math.cpp* au complet est donc :

Code : C++ - Le fichier *math.cpp*

```
#include "math.h"

int ajouteDeux(int nombreRecu)
{
    int valeur(nombreRecu + 2);

    return valeur;
}
```

Le fichier d'en-tête

Si vous regardez le fichier qui a été créé, il n'est pas vide ! 🤖 Il contient trois lignes mystérieuses :

Code : C++ - Les gardes anti-inclusions multiples

```
#ifndef MATH_H_INCLUDED
#define MATH_H_INCLUDED

#endif // MATH_H_INCLUDED
```

Ces lignes sont là pour empêcher le compilateur d'inclure plusieurs fois ce fichier. Le compilateur n'est parfois pas très malin et risque de tourner en rond. Cette astuce évite de se retrouver dans cette situation. Il ne faut donc pas toucher ces lignes et surtout, écrire tout le code **entre la deuxième et la troisième**.

Le texte en majuscule sera différent pour chaque fichier. C'est le texte qui apparaît dans le champ que nous n'avons pas modifié lors de la création du fichier.



Si vous n'utilisez pas Code::Blocks, vous n'aurez peut-être pas automatiquement ces lignes dans vos fichiers. Il faut alors les ajouter à la main. Le mot en majuscule doit être le même sur les 3 lignes où il apparaît. Et chaque fichier doit utiliser un mot différent.

Dans ce fichier, il faut mettre ce qu'on appelle le **prototype** de la fonction. C'est la première ligne de la fonction. Celle qui vient avant les accolades. On prend cette ligne et on ajoute un point-virgule à la fin.

C'est donc très court. Voici ce que l'on obtient pour notre fonction :

Code : C++ - Contenu du fichier d'en-tête

```
#ifndef MATH_H_INCLUDED
#define MATH_H_INCLUDED

int ajouteDeux(int nombreRecu);

#endif // MATH_H_INCLUDED
```



N'oubliez pas le point-virgule ici !

Et c'est tout. 🤔 Il ne nous reste qu'une seule chose à faire : inclure tout ça dans le fichier *main.cpp*. Si on ne le fait pas, le compilateur ne saura pas où trouver la fonction `ajouteDeux()` lorsqu'on essaiera de l'utiliser. Il faut donc ajouter la ligne

Code : C++

```
#include "math.h"
```

au début de notre programme. Ce qui donne :

Code : C++

```
#include <iostream>
#include "math.h"
using namespace std;

int main()
{
    int a(2), b(2);
    cout << "Valeur de a : " << a << endl;
    cout << "Valeur de b : " << b << endl;
    b = ajouteDeux(a); //Appel de la fonction
    cout << "Valeur de a : " << a << endl;
    cout << "Valeur de b : " << b << endl;

    return 0;
}
```



On inclut toujours le fichier d'en-tête (.h). Jamais le fichier source (.cpp).

Et voilà ! Nous avons maintenant réellement des briques séparées utilisables dans plusieurs programmes. Si vous voulez utiliser la fonction `ajouteDeux()` dans un autre projet, il vous suffira de copier les fichiers *math.cpp* et *math.h*. 🧙‍♂️



On peut bien sûr mettre plusieurs fonctions par fichier. On les regroupe généralement par catégorie. Les fonctions mathématiques d'un côté, les fonctions pour l'affichage d'un menu dans un autre fichier et celle pour faire se déplacer un personnage de jeu vidéo dans un troisième. Programmer, c'est aussi être organisé. 🤖

Documenter son code

Avant de terminer ce chapitre, je veux juste vous présenter un point qui peut sembler futile. On vous l'a dit dès le début, il est

important de mettre des commentaires dans son programme pour comprendre ce qu'il fait. C'est particulièrement vrai pour les fonctions puisque vous allez peut-être utiliser des fonctions écrites par d'autres programmeurs. Il est donc important de savoir ce que font ces fonctions sans avoir besoin de lire tout le code ! (Rappelez-vous, le programmeur est fainéant...)

Comme il y a de la place dans les fichiers d'en-tête, on en profite généralement pour décrire ce que font les fonctions. Il y a généralement trois choses décrites :

1. Ce que fait la fonction.
2. La liste des ses arguments.
3. La valeur retournée.

Plutôt qu'un long discours, voici ce qu'on pourrait écrire pour notre fonction `ajouteDeux()` :

Code : C++ - Le fichier `math.h` avec des commentaires

```
#ifndef MATH_H_INCLUDED
#define MATH_H_INCLUDED

/*
 * Fonction qui ajoute 2 au nombre reçu en argument
 * - nombreRecu : Le nombre auquel la fonction ajoute 2
 * Valeur retournée : nombreRecu + 2
 */
int ajouteDeux(int nombreRecu);

#endif // MATH_H_INCLUDED
```

Bien sûr, dans ce cas, le descriptif est très simple. Mais c'est une habitude qu'il faut prendre. C'est d'ailleurs tellement courant de mettre des commentaires dans les fichiers `.h` qu'il existe des systèmes quasi-automatiques qui génèrent des sites web ou des livres à partir de ces commentaires. 🤖

Le célèbre système `doxygen` utilise par exemple la notation suivante :

Code : C++

```
/**
 * \brief Fonction qui ajoute 2 au nombre reçu en argument
 * \param nombreRecu Le nombre auquel la fonction ajoute 2
 * \return nombreRecu + 2
 */
int ajouteDeux(int nombreRecu);
```

Pour l'instant cela peut vous paraître un peu inutile, mais vous verrez dans la partie III de ce cours qu'avoir une bonne documentation est essentiel. A vous de choisir la notation que vous préférez.

Des valeurs par défaut pour les arguments

Les arguments de fonctions, vous commencez à connaître. Je vous en parle depuis le début du chapitre. Lorsque une fonction a trois arguments, il faut lui fournir trois valeurs pour qu'elle puisse fonctionner.

Et bien, je vais vous montrer que ce n'est pas toujours le cas.

Voyons tout ça avec la fonction suivante :

Code : C++

```
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
```



```
total += minutes * 60;
total += secondes;

return total;
}
```

Cette fonction calcule le nombre de secondes en additionnant les heures, minutes et secondes qu'on lui envoie. Rien de bien compliqué ! 😊

Les variables `heures`, `minutes` et `secondes` sont les **paramètres** de la fonction `nombreDeSecondes()`. Ce sont des valeurs qu'elle reçoit, celles avec lesquelles elle va travailler.

Mais ça, vous le savez déjà. 😊

Les valeurs par défaut

La nouveauté, c'est qu'on peut donner des valeurs par défaut à certains paramètres de nos fonctions. Ainsi, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres lorsqu'on appelle une fonction !

Pour bien voir comment on doit procéder, on va regarder le code complet. J'aimerais que vous le copiiez dans votre IDE pour faire les tests en même temps que moi :

Code : C++

```
#include <iostream>

using namespace std;

// Prototype de la fonction
int nombreDeSecondes(int heures, int minutes, int secondes);

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;

    return 0;
}

// Définition de la fonction
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```

Ce code donne le résultat suivant :

Code : Console

```
4225
```

Sachant qu'1 heure = 3600s, 10 minutes = 600s, 25 secondes =... 25s, le résultat est logique car $3600 + 600 + 25 = 4225$.



Bref, tout va bien.

Maintenant supposons que l'on veuille rendre certains paramètres facultatifs, par exemple parce qu'on utilise en pratique plus souvent les heures que le reste.

On va devoir modifier le prototype de la fonction (et non sa définition, attention).

Indiquez la valeur par défaut que vous voulez donner aux paramètres si on ne les a pas renseignés lors de l'appel de la fonction :

Code : C++

```
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
```

Dans cet exemple, seul le paramètre heures sera obligatoire, les deux autres étant désormais facultatifs. Si on ne renseigne pas les minutes et les secondes, les variables vaudront alors 0 dans la fonction.

Voici le code complet que vous devriez avoir sous les yeux :

Code : C++

```
#include <iostream>

using namespace std;

// Prototype avec les valeurs par défaut
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;

    return 0;
}

// Définition de la fonction, SANS les valeurs par défaut
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```



Si vous avez lu attentivement ce code, vous avez dû vous rendre compte de quelque chose : les valeurs par défaut sont spécifiées uniquement dans le prototype, PAS dans la définition de la fonction ! Si votre code est découpé en plusieurs fichiers, alors il ne faut spécifier les valeurs par défaut que dans le fichier d'en-tête `.h`. On se fait souvent avoir, je vous préviens... 🤔

Si vous vous trompez, le compilateur vous indiquera une erreur à la ligne de la définition de la fonction.

Bon, ce code ne change pas beaucoup du précédent. A part les valeurs par défaut dans le prototype, rien n'a été modifié (et le résultat à l'écran sera toujours le même).

La nouveauté maintenant, c'est qu'on peut supprimer des paramètres lors de l'appel de la fonction (ici dans le `main()`). On peut par exemple écrire :

Code : C++

```
cout << nombreDeSecondes(1) << endl;
```

Le compilateur lit les paramètres de gauche à droite. Comme il n'y en a qu'un et que seules les heures sont obligatoires, il devine que la valeur "1" correspond à un nombre d'heures.

Le résultat à l'écran sera le suivant :

Code : Console

```
3600
```

Mieux encore, vous pouvez indiquer juste les heures et les minutes si vous le désirez :

Code : C++

```
cout << nombreDeSecondes(1, 10) << endl;
```

Code : Console

```
4200
```

Tant que vous indiquez au moins les paramètres obligatoires, il n'y a pas de problème. 😊

Cas particuliers, attention danger

Bon, mine de rien il y a quand même quelques pièges, ce n'est pas si simple que ça ! 😊

On va voir ces pièges sous la forme de questions / réponses :



Et si je veux envoyer à la fonction juste les heures et les secondes, mais pas les minutes ?

Tel quel, c'est impossible. En effet, je vous l'ai dit plus haut, le compilateur va analyser les paramètres de gauche à droite. Le premier correspondra forcément aux heures, le second aux minutes et le troisième aux secondes.

Vous ne pouvez PAS écrire :

Code : C++

```
cout << nombreDeSecondes(1, , 25) << endl;
```

C'est interdit. Si vous le faites, le compilateur vous fera comprendre qu'il n'apprécie guère vos manœuvres. C'est comme ça : en C++, on ne peut pas "sauter" des paramètres, même s'ils sont facultatifs. Si vous voulez indiquer le premier et le dernier paramètre, il vous faudra obligatoirement spécifier ceux du milieu. On devra donc écrire :

Code : C++

```
cout << nombreDeSecondes(1, 0, 25) << endl;
```



Est-ce que je peux rendre juste les heures facultatives, et rendre les minutes et secondes obligatoires ?

Si le prototype est défini dans le même ordre que tout à l'heure : non.

Les paramètres facultatifs doivent obligatoirement se trouver à la fin (à droite).

Ce code ne compilera donc pas :

Code : C++

```
int nombreDeSecondes(int heures = 0, int minutes, int secondes);  
// Erreur, les paramètres par  
défaut doivent être à droite
```

La solution, pour régler ce problème, consiste à placer le paramètre *heures* à la fin :

Code : C++

```
int nombreDeSecondes(int secondes, int minutes, int heures = 0);  
// OK
```



Est-ce que je peux rendre tous mes paramètres facultatifs ?

Oui, ça ne pose pas de problème :

Code : C++

```
int nombreDeSecondes(int heures = 0, int minutes = 0, int secondes =  
0);
```

Dans ce cas, l'appel de la fonction pourra être fait comme ceci :

Code : C++

```
cout << nombreDeSecondes() << endl;
```

Le résultat retourné sera bien entendu 0 dans notre cas. 🤖

Règles à retenir

En résumé, il y a 2 règles que vous devez retenir pour les valeurs par défaut :

- Seul le prototype doit contenir les valeurs par défaut (pas la définition de la fonction).

- Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres ("à droite").

Souvenez-vous qu'il est très important de découper son programme en fonctions. Dans certaines entreprises, on oblige les gens à faire des fonctions dès qu'un morceau de code dépasse une hauteur d'écran ! C'est peut-être un peu extrême, mais force les programmeurs à avoir la bonne attitude, découper, découper et découper encore. 😊

Au terme de ce chapitre, vous connaissez les variables, les branchements et les fonctions. Vous connaissez donc presque tous les concepts de base. 😊 Dans le chapitre suivant, nous allons découvrir un nouveau type de variables.

Les tableaux

Vous êtes encore là ?

Bien ! Vous êtes maintenant familiers avec la notion de variable et vous avez pu voir à quel point les utiliser est important. Il est temps maintenant d'aborder des types de variables un petit peu plus complexes : les tableaux.

Dans de très nombreux programmes, on a besoin d'avoir plusieurs variables du même type et qui jouent quasiment le même rôle. Pensez par exemple à la liste des utilisateurs d'un site web. Cela représente une grande quantité de variables de type `string`. Ou les dix meilleurs scores de votre jeu que l'on stockera dans différentes cases mémoires, toutes de type `int`. Le C++, comme presque tous les langages de programmation, propose un moyen simple de regrouper des données identiques en un seul paquet. Et comme l'indique le titre du chapitre, on appelle ces regroupements de variables **des tableaux**.

Dans ce chapitre, je vais vous apprendre à manipuler deux sortes de tableaux. Ceux dont la taille est connue à l'avance, comme pour la liste des dix meilleurs scores. Et ceux dont la taille peut varier en permanence comme la liste des visiteurs d'un site web qui, généralement, ne cesse de grandir.

Vous vous en doutez certainement, les tableaux dont la taille est fixée à l'avance sont plus faciles à utiliser et c'est donc par eux que nous allons commencer.

Les tableaux statiques

Je vous ai parlé dans l'introduction de l'intérêt des tableaux dans le cas où l'on a plusieurs variables du même type à stocker. Voyons cela avec un exemple bien connu, la liste des meilleurs scores du jeu révolutionnaire que vous allez créer un jour. 😎

Un exemple d'utilisation

Si vous voulez afficher la liste des 5 meilleurs scores des joueurs, il va vous falloir en réalité deux listes. La liste des noms de joueurs et la liste des scores qu'ils ont obtenus. Il va donc falloir déclarer 10 variables pour mettre toutes ces informations dans la mémoire de l'ordinateur. On aura par exemple :

Code : C++ - Les variables du 'Hall of fame'

```
string nomMeilleurJoueur1("Nanoc");
string nomMeilleurJoueur2("M@teo21");
string nomMeilleurJoueur3("Albert Einstein");
string nomMeilleurJoueur4("Isaac Newton");
string nomMeilleurJoueur5("Archimede");

int meilleurScore1(118218);
int meilleurScore2(100432);
int meilleurScore3(87347);
int meilleurScore4(64523);
int meilleurScore5(31415);
```

Et pour afficher tout ça, il va aussi falloir pas mal de travail.

Code : C++ - Affichage du 'Hall of fame'

```
cout << "1) " << nomMeilleurJoueur1 << " " << meilleurScore1 << endl;
cout << "2) " << nomMeilleurJoueur2 << " " << meilleurScore2 << endl;
cout << "3) " << nomMeilleurJoueur3 << " " << meilleurScore3 << endl;
cout << "4) " << nomMeilleurJoueur4 << " " << meilleurScore4 << endl;
cout << "5) " << nomMeilleurJoueur5 << " " << meilleurScore5 << endl;
```

Ce qui fait énormément de lignes ! Imaginez maintenant que vous vouliez afficher les 100 meilleurs scores et pas seulement les 5

meilleurs. Ça serait terrible. Il vous faudrait déclarer 200 variables et écrire 100 lignes quasiment identiques pour l'affichage ! Autant arrêter tout de suite, c'est beaucoup trop de travail. 😓

C'est là qu'interviennent les tableaux. Nous allons pouvoir déclarer les 100 meilleures scores et les noms des 100 meilleurs joueurs d'un seul coup. On va créer **une seule** case dans la mémoire qui aura de la place pour contenir les 100 `int` qu'il nous faut et une deuxième pour contenir les 100 `string`. Magique non ?



Il faut quand même que les variables aient un lien entre elles pour que l'utilisation d'un tableau soit justifiée. Mettre dans un même tableau l'âge de votre chien et le nombre d'internautes connectés n'est pas correct. Même si ces deux variables sont des `int`.

Dans cet exemple, nous avons besoin de 100 variables. C'est-à-dire 100 places dans le tableau. C'est ce qu'on appelle, en termes techniques, la **taille** du tableau. Si la taille du tableau reste inchangée et est fixée dans le code source, alors on parle d'un **tableau statique**. Parfait ! C'est ce dont nous avons besoin pour notre liste des 100 meilleurs scores.

Déclarer un tableau statique

Comme toujours en C++, une variable est composée d'un nom et d'un type. Comme les tableaux sont des variables, cette règle reste valable. Il faut juste ajouter une propriété supplémentaire, la taille du tableau. Autrement dit, le nombre de compartiments que notre case mémoire va pouvoir contenir.

La déclaration d'un tableau est très similaire à celle d'une variable :

TYPE **NOM** [**TAILLE**] ; Déclaration d'un tableau statique

On indique le type, puis le nom choisi et enfin la taille du tableau entre crochets. Voyons ça avec un exemple.

Code : C++ - Votre premier tableau

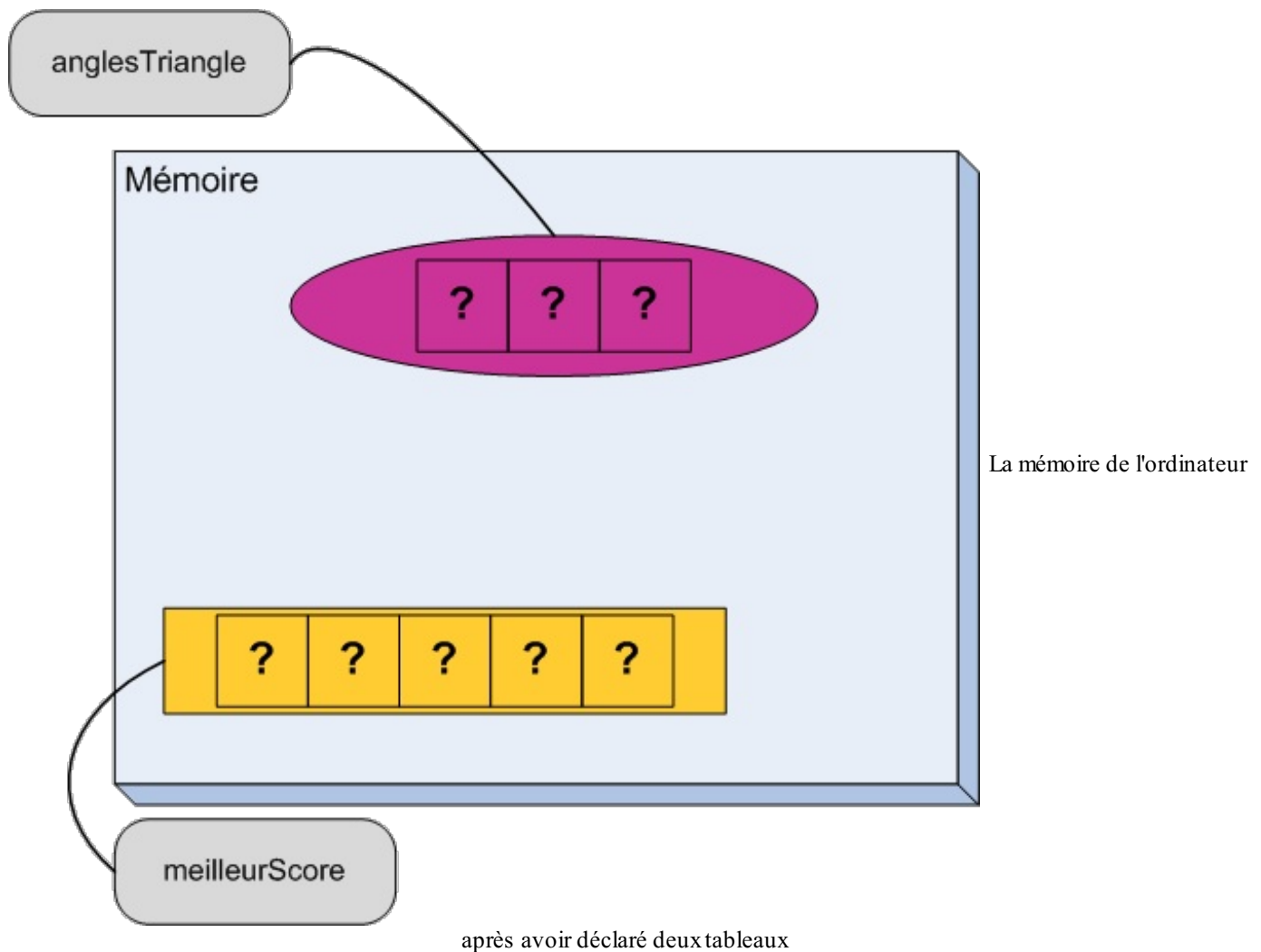
```
#include <iostream>
using namespace std;

int main()
{
    int meilleurScore[5];    //Declare un tableau de 5 entiers

    double anglesTriangle[3]; //Declare un tableau de 3 double

    return 0;
}
```

Voyons à quoi ressemble la mémoire avec un de nos schémas habituels.



On retrouve nos deux zones mémoires avec leurs étiquettes, mais cette fois, chaque zone est découpée en cases. Trois cases pour le tableau `anglesTriangle` et cinq cases pour le tableau `meilleurScore`. Pour l'instant toutes ces cases ne sont pas initialisées. Leur contenu est donc quelconque.

Il est également possible de déclarer un tableau en utilisant comme taille une **constante** de type `int` ou `unsigned int`. On indique simplement le nom de la constante entre les crochets plutôt qu'un nombre.

Code : C++

```
int const tailleTableau(20); //La taille du tableau
double anglesIcosagone[tailleTableau];
```



Il faut **impérativement** utiliser une **constante** comme taille du tableau.

Je vous conseille de toujours utiliser des constantes comme taille de vos tableaux plutôt que d'indiquer directement la taille entre les crochets. C'est une bonne habitude à prendre.

Bon. On a de la place dans la mémoire. Il ne nous reste plus qu'à l'utiliser. 😊

Accéder aux éléments d'un tableau

Chaque case d'un tableau peut être utilisée comme n'importe quelle autre variable. Il n'y a aucune différence. Il faut juste y

accéder d'une manière un peu spéciale. Il faut indiquer le nom du tableau et le numéro de la case. Dans le tableau `meilleurScore`, on a accès à cinq variables. La première case de `meilleurScore`, la deuxième, etc, jusqu'à la cinquième.

Pour accéder à une case on utilise la syntaxe `nomDuTableau[numeroDeLaCase]`. Il y a juste une petite subtilité, la première case possède le numéro 0 et pas 1. Tout est en quelque sorte décalé de 1. Pour accéder à la 3^e case de `meilleurScore` et y écrire une valeur, il faudra donc écrire :

Code : C++

```
meilleurScore[2] = 5;
```

En effet, **3 – 1 = 2**, la 3^e case possède le numéro 2. Si je veux remplir mon tableau des meilleurs scores comme dans l'exemple initial, je peux donc écrire :

Code : C++ - Remplissage d'un tableau

```
int const nombreMeilleursScores(5); //La taille du tableau

int meilleursScores[nombreMeilleursScores]; //Declaration du
tableau

meilleursScores[0] = 118218; //Remplissage de la premiere case
meilleursScores[1] = 100432; //Remplissage de la deuxieme case
meilleursScores[2] = 87347; //Remplissage de la troisieme case
meilleursScores[3] = 64523; //Remplissage de la quatrieme case
meilleursScores[4] = 31415; //Remplissage de la cinquieme case
```



Comme tous les numéros de cases sont décalés, la dernière case a le numéro 4 et pas 5 !

Parcourir un tableau

Le gros point fort des tableaux, c'est qu'on peut les parcourir en utilisant une boucle. On peut ainsi effectuer une action sur chacune des cases d'un tableau l'une après l'autre. Par exemple afficher le contenu des cases.

On connaît à priori le nombre de cases du tableau, on peut donc utiliser une boucle **for**. Nous allons pouvoir utiliser la variable `i` de la boucle pour accéder au *i*ème élément du tableau. C'est fou, on dirait que c'est fait pour !

Code : C++ - Parcourir un tableau

```
int const nombreMeilleursScores(5); //La taille du tableau
int meilleursScores[nombreMeilleursScores]; //Declaration du
tableau

meilleursScores[0] = 118218; //Remplissage de la premiere case
meilleursScores[1] = 100432; //Remplissage de la deuxiemecase
meilleursScores[2] = 87347; //Remplissage de la troisieme case
meilleursScores[3] = 64523; //Remplissage de la quatrieme case
meilleursScores[4] = 31415; //Remplissage de la cinquieme case

for(int i(0); i<nombreMeilleursScores; ++i)
{
    cout << meilleursScores[i] << endl;
}
```

La variable `i` va prendre successivement les valeurs 0,1,2,3 et 4. Ce qui veut dire que ce seront les valeurs de `meilleursScores[0]` puis `meilleursScores[1]` etc. qui seront envoyées dans `cout`.



Il faut faire très attention à ne pas dépasser la taille du tableau dans la boucle. Sinon, vous aurez droit à un plantage de votre programme. La dernière case dans cet exemple a le numéro `nombreMeilleursScores` **moins un**. Les valeurs autorisées de `i` sont tous les entiers entre 0 et `nombreMeilleursScores` **moins un** compris.

Vous allez voir, le couple tableau - boucle `for` va devenir votre nouveau meilleur ami. En tout cas je l'espère. 😊 C'est un outil très puissant.

Un petit exemple

Allez, je vous propose un petit exemple un petit peu plus complexe. Nous allons utiliser le C++ pour calculer la moyenne de vos notes de l'année. Je vous propose de mettre toutes vos notes dans un tableau et d'utiliser une boucle `for` pour le calcul de la moyenne.

Vous voyez comment faire ? Parfait ! Je vous regarde. 🤖

Bon. Ok. J'accepte de vous aider. Voyons donc tout ça étape par étape. Premièrement, il nous faut un tableau pour stocker les notes. Comme ce sont des nombres à virgule, il nous faut des `double`.

Code : C++

```
int const nombreNotes(6);  
double notes[nombreNotes];
```

La deuxième étape consiste à remplir ce tableau avec vos notes. J'espère que vous savez encore comment faire !

Code : C++

```
int const nombreNotes(6);  
double notes[nombreNotes];  
  
notes[0] = 12.5;  
notes[1] = 19.5; //Bieeeeeen !  
notes[2] = 6.;   //Pas bien !  
notes[3] = 12;  
notes[4] = 14.5;  
notes[5] = 15;
```



Je me répète, mais c'est important. La première case du tableau a le numéro 0. La deuxième a le numéro 1 et ainsi de suite.

Pour calculer la moyenne, il nous faut additionner toutes les notes et ensuite diviser par le nombre de notes. Nous connaissons déjà le nombre de notes, puisque nous avons la constante `nombreNotes`. Il ne reste donc qu'à déclarer une variable pour contenir la moyenne.

Le calcul de la somme s'effectue dans une boucle `for` qui va parcourir toutes les cases du tableau.

Code : C++

```
double moyenne(0);  
for(int i(0); i<nombreNotes; ++i)  
{  
    moyenne += notes[i]; //On additionne toutes les notes
```

```
}  
//En arrivant ici, la variable moyenne contient la somme des notes  
(79.5)  
//Il ne reste donc qu'a diviser par le nombre de notes  
moyenne /= nombreNotes;
```

Avec une petite ligne pour l'affichage de la valeur, on obtient le résultat voulu. Un programme qui calcule la moyenne de vos notes.

Code : C++ - Calcul de la moyenne des notes

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int const nombreNotes(6);  
    double notes[nombreNotes];  
  
    notes[0] = 12.5;  
    notes[1] = 19.5; //Bieeeeeen !  
    notes[2] = 6.;   //Pas bien !  
    notes[3] = 12;  
    notes[4] = 14.5;  
    notes[5] = 15;  
  
    double moyenne(0);  
    for(int i(0); i<nombreNotes; ++i)  
    {  
        moyenne += notes[i]; //On additionne toutes les notes  
    }  
    //En arrivant ici, la variable moyenne contient la somme des  
    notes (79.5)  
    //Il ne reste donc qu'a diviser par le nombre de notes  
    moyenne /= nombreNotes;  
  
    cout << "Votre moyenne est : " << moyenne << endl;  
  
    return 0;  
}
```

Voyons ce que ça donne quand on l'exécute.

Code : Console

```
Votre moyenne est : 13.25
```

Et ça marche ! Vous n'en doutiez pas j'espère. 😊

Les tableaux et les fonctions

Ah ! les fonctions. Vous n'avez pas oublié ce que c'est j'espère. Il faut quand même que je vous en reparle un peu. Comme vous allez le voir, les tableaux et les fonctions ne sont pas les meilleurs amis du monde.

La première restriction est qu'on ne peut pas écrire une fonction qui renvoie un tableau statique. C'est impossible. 😞

La deuxième restriction est qu'un tableau statique est **toujours passé par référence**. Et il n'y a pas besoin d'utiliser l'esperluette (&). C'est fait automatiquement. Cela veut dire que lorsqu'on passe un tableau à une fonction, cette dernière peut le modifier.

Voici donc une fonction qui reçoit un tableau en argument.

Code : C++

```
void fonction(double tableau[])
{
    //...
}
```



Il ne faut rien mettre entre les crochets.

Mais ce n'est pas tout ! Très souvent, on va vouloir parcourir le tableau, avec une boucle **for** par exemple. Il nous manque une information cruciale. Vous voyez laquelle ?

La taille ! A l'intérieur de la fonction précédente, il n'y a aucun moyen de connaître la taille du tableau ! Il faut donc **impérativement ajouter un deuxième argument contenant la taille**. Ce qui donne :

Code : C++

```
void fonction(double tableau[], int tailleTableau)
{
    //...
}
```

Oui, je sais c'est ennuyeux. Mais il ne faut pas vous en prendre à moi. Je n'ai pas créé le langage. 🙄

Pour vous entraîner, je vous propose d'écrire une fonction `moyenne()` qui calcule la moyenne des valeurs d'un tableau.

Voici ma version :

Secret (cliquez pour afficher)

Code : C++

```
/*
 * Fonction qui calcule la moyenne des éléments d'un tableau
 * - tableau: Le tableau dont on veut la moyenne
 * - tailleTableau: La taille du tableau
 */
double moyenne(double tableau[], int tailleTableau)
{
    double moyenne(0);
    for(int i(0); i<tailleTableau; ++i)
    {
        moyenne += tableau[i];    //On additionne toutes les valeurs
    }
    moyenne /= tailleTableau;

    return moyenne;
}
```

Bon bon, assez parlé de ces tableaux. Passons à la suite.

Les tableaux dynamiques

Je vous avais dit que nous allions parler de deux sortes de tableaux différents. Ceux dont la taille est fixée et ceux dont la taille peut varier, les **tableaux dynamiques**. Certaines choses sont identiques, ce qui va nous permettre de ne pas tout répéter.

Déclarer un tableau dynamique

La première différence se situe au tout début de votre programme. Il faut ajouter la ligne `#include <vector>` pour utiliser ces tableaux.



A cause de cette ligne, on parle souvent de `vector` à la place de tableau dynamique. J'utiliserai ce terme parfois dans la suite.

La deuxième grosse différence se situe dans la manière de déclarer un tableau. On utilise la syntaxe

vector<TYPE > NOM (TAILLE) ;

Par exemple pour un tableau de 5 entiers, on écrit :

Code : C++

```
#include <iostream>
#include <vector> //Ne pas oublier !
using namespace std;

int main()
{
    vector<int> tableau(5);

    return 0;
}
```

Il faut remarquer trois choses.

1. Le type n'est pas le premier mot de la ligne comme pour toutes les autres variables.
2. On utilise une notation bizarre avec un chevron ouvrant et un chevron fermant.
3. On écrit la taille entre des parenthèses et pas entre crochets.

Ce qui veut dire que ça ne ressemble pas vraiment aux tableaux statiques. 🤖 Mais vous allez voir, pour parcourir le tableau, le principe est similaire.

Mais avant cela, il y a deux astuces bien pratiques à savoir.

On peut directement remplir toutes les cases du tableau en ajoutant un deuxième argument entre les parenthèses.

Code : C++

```
vector<int> tableau(5, 3); //Cree un tableau de 5 entiers valant
tous 3
vector<string> listeNoms(12, "Sans nom"); //Cree un tableau de 12
strings valant toutes "Sans nom"
```

On peut déclarer un tableau sans cases en ne mettant pas de parenthèses du tout.

Code : C++

```
vector<double> tableau; //Cree un tableau de 0 nombres à virgules
```



Euh... A quoi ça sert un tableau vide ?

Hehe, rappelez-vous que ce sont des tableaux dont la taille peut varier. On peut donc ajouter des cases par la suite. Attendez un peu et vous saurez tout. 😊

Accéder aux éléments d'un tableau

La déclaration était très différente des tableaux statiques. Par contre là, c'est exactement identique. On utilise à nouveau les crochets et la première case possède aussi le numéro 0.

On peut donc récrire encore une fois l'exemple de la sous-partie précédente avec un vector.

Code : C++ - Remplissage d'un tableau

```
int const nombreMeilleursScores(5); //La taille du tableau

vector<int> meilleursScores(nombreMeilleursScores); //Déclaration
du tableau

meilleursScores[0] = 118218; //Remplissage de la première case
meilleursScores[1] = 100432; //Remplissage de la deuxième case
meilleursScores[2] = 87347; //Remplissage de la troisième case
meilleursScores[3] = 64523; //Remplissage de la quatrième case
meilleursScores[4] = 31415; //Remplissage de la cinquième case
```

Là, je crois qu'on ne peut pas faire plus facile. 🧙

Changer la taille

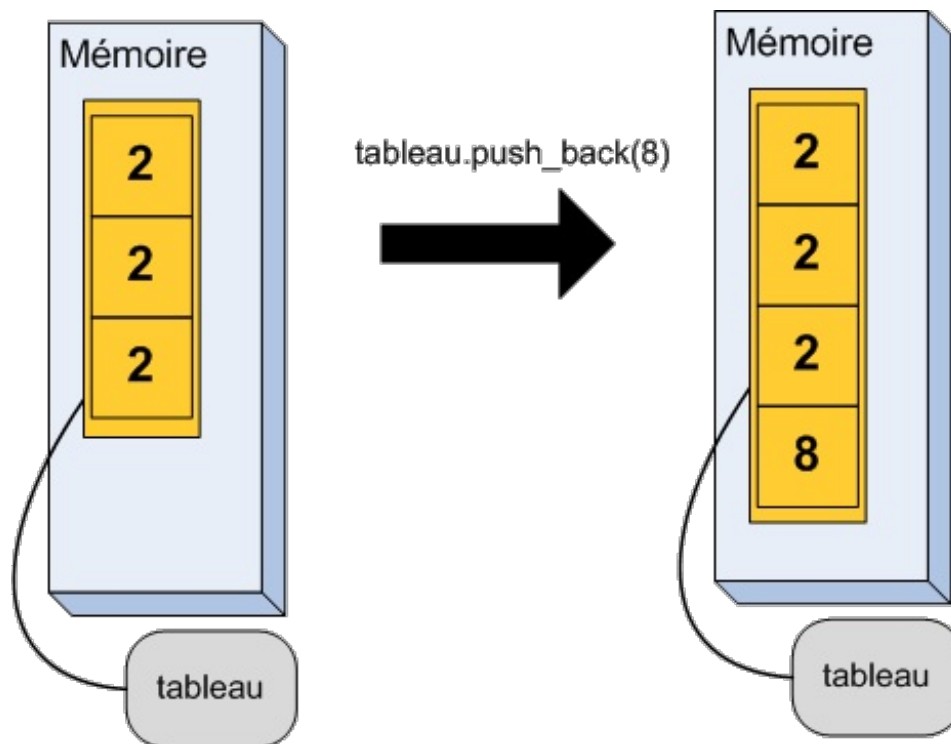
Entrons maintenant dans le vif du sujet. Faire varier la taille d'un tableau. Commençons par ajouter des cases à la fin d'un tableau.

Il faut utiliser `push_back()`. On écrit le nom du tableau, suivi d'un point et du mot `push_back` avec entre parenthèses la valeur qui va remplir la nouvelle case.

Code : C++

```
vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2
tableau.push_back(8); //On ajoute une 4ème case au tableau. Cette
case contient la valeur 8
```

Voyons ce qui se passe dans la mémoire de plus près.



Une case supplémentaire a été ajoutée au bout du tableau. Tout se fait de manière automatique. C'est fou ce que ça peut être intelligent un ordinateur parfois. 😊

Et bien sûr on peut ajouter beaucoup de cases à la suite les unes des autres.

Code : C++

```
vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2
tableau.push_back(8); //On ajoute une 4ème case au tableau. Cette
case contient la valeur 8
tableau.push_back(7); //On ajoute une 5ème case qui contient le
chiffre 7.
tableau.push_back(14); //Et encore une avec le nombre 14 cette
fois.

//Le tableau contient maintenant les nombres : 2 2 2 8 7 14
```



Et ils ne peuvent que grandir les vectors ?

Non ! Bien sûr que non. Les créateurs du C++ ont pensé à tout. 😎

On peut supprimer la dernière case d'un tableau en utilisant la fonction `pop_back()` de la même manière que `push_back()`. Sauf qu'il n'y a rien à mettre entre les parenthèses.

Code : C++

```
vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2
tableau.pop_back(); //Et hop ! Plus que 2 cases.
tableau.pop_back(); //Et hop ! Plus qu'une case.
```



Attention tout de même à ne pas trop supprimer de cases ! Un tableau ne peut pas contenir moins de 0 éléments.



Je crois que je n'ai pas besoin d'en dire plus sur ce sujet.

Il nous reste quand même un petit problème à régler. Comme la taille peut changer, on ne sait pas combien d'éléments un tableau contient de manière certaine. Heureusement, il y a une fonction pour ça. C'est la fonction `size()`. En faisant `tableau.size()`, on récupère un entier correspondant au nombre d'éléments de tableau.

Code : C++

```
vector<int> tableau(5,4); //Un tableau de 5 entiers valant tous 4
int const taille(tableau.size()); //Une variable pour contenir la
    taille du tableau
                                //La taille peut varier mais la
    valeur de cette variable ne changera pas.
                                //On utilise donc une constante.
                                //A partir d'ici, la constante
    'taille' vaut donc 5
```

Retour sur l'exercice

Je crois que le mieux pour se mettre tout ça en tête, est de reprendre l'exercice du calcul des moyennes mais en le refaisant à la "sauce vector".

Je vous laisse essayer. Si vous n'y arrivez pas, voici ma solution :

Code : C++ - Calcul de moyenne en utilisant vector

```
#include <iostream>
#include <vector> //Ne pas oublier !!
using namespace std;

int main()
{
    vector<double> notes; //Un tableau vide

    notes.push_back(12.5); //On ajoute des cases avec les notes
    notes.push_back(19.5);
    notes.push_back(6);
    notes.push_back(12);
    notes.push_back(14.5);
    notes.push_back(15);

    double moyenne(0);
    for(int i(0); i<notes.size(); ++i) //On utilise notes.size()
    pour la limite de notre boucle
    {
        moyenne += notes[i]; //On additionne toutes les notes
    }

    moyenne /= notes.size(); //On utilise a nouveau notes.size()
    pour obtenir le nombre de notes

    cout << "Votre moyenne est : " << moyenne << endl;

    return 0;
}
```

Wow ! C'est assez différent en fait. 😊

On a écrit deux programmes qui font exactement la même chose de deux manières différentes. C'est très courant. Il y a presque toujours plusieurs manières de faire les choses. Chacun choisit celle qu'il préfère.

Les vector et les fonctions

Passer un tableau dynamique en argument à une fonction est beaucoup plus simple que pour les tableaux statiques. Comme pour n'importe quel autre type, il suffit de mettre `vector<type>` en argument. Et c'est tout. Grâce à la fonction `size()`, il n'y a même pas besoin d'ajouter un deuxième argument pour la taille du tableau ! 🎉

Ce qui donne tout simplement :

Code : C++ - Passer un vector en argument d'une fonction

```
void fonction(vector<int> a)    //Une fonction recevant un tableau
d'entiers en argument
{
    //...
}
```

Simple non ? Mais on peut quand même faire mieux. Je vous ai parlé dans le [chapitre précédent](#) du passage par référence constante pour optimiser la copie. En effet, si le tableau contient beaucoup d'éléments, le copier prendra du temps. Il vaut donc mieux utiliser cette astuce, ce qui donne :

Code : C++ - Passer un vector en argument d'une fonction

```
void fonction(vector<int> const& a)    //Une fonction recevant un
tableau d'entiers en argument
{
    //...
}
```

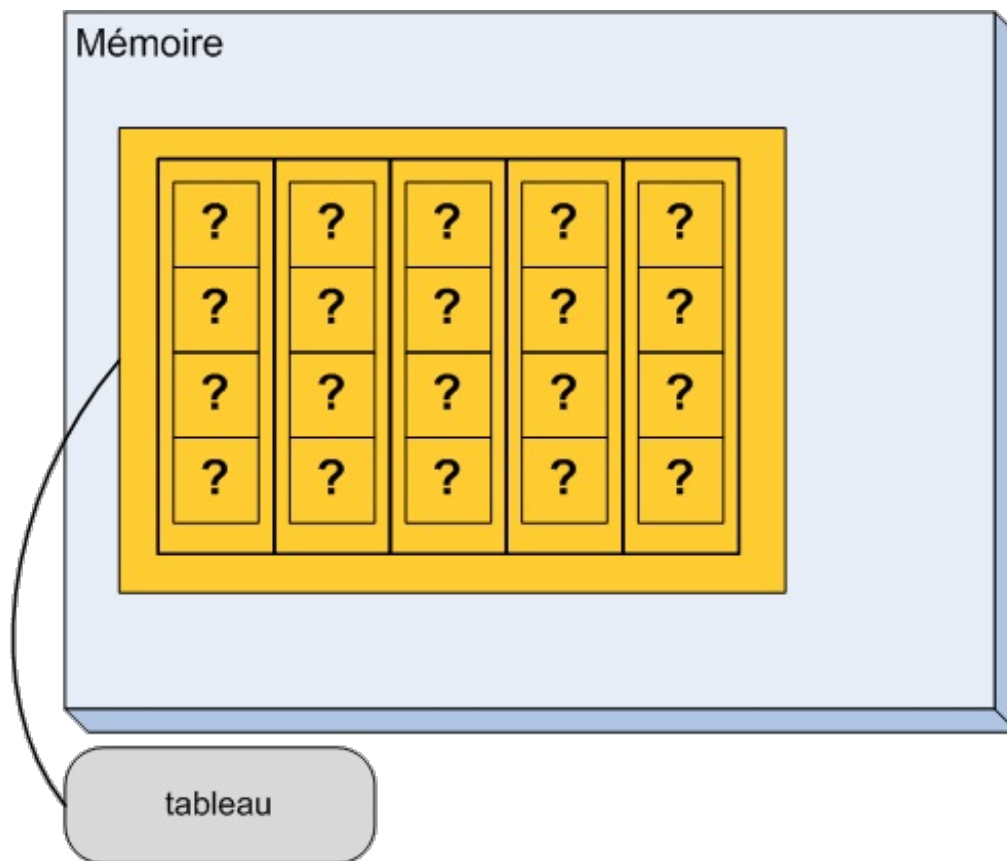


Le tableau dynamique ne peut pas être modifié dans ce cas. Pour changer le contenu du tableau, il faut utiliser un passage par référence tout simple (sans le `const` donc).

Les tableaux multi-dimensionnels

Je vous ai dit en début de chapitre que l'on pouvait créer des tableaux de n'importe quoi. Des tableaux d'entiers, des tableaux de strings, et ainsi de suite. On peut donc créer des tableaux de ... tableaux ! 🤖

Je vous vois d'ici froncer les sourcils et vous demander à quoi cela peut bien servir. Une fois n'est pas coutume, je vous propose de commencer par visualiser la mémoire. Vous verrez peut-être l'intérêt de ce concept pour le moins bizarre.



La grosse case jaune représente, comme à chaque fois, une variable. Cette fois, c'est un tableau de 5 éléments dont j'ai représenté les cases en utilisant des lignes épaisses. A l'intérieur de chacune des cases, on trouve un petit tableau de 4 éléments dont on ne connaît pas la valeur. Pffiu... 🤔

Mais si vous regardez attentivement les points d'interrogation, vous pouvez voir une grille régulière ! Un tableau de tableau est donc une grille de variables. Et là, je pense que vous trouvez ça tout de suite moins bizarre.



On parle parfois de **tableaux multi-dimensionnels** plutôt que de grilles. C'est pour souligner le fait que les variables sont arrangées selon des axes **X** et **Y** et pas juste selon un seul axe.

Déclaration d'un tableau multi-dimensionnel

Pour déclarer un tel tableau, il faut indiquer les dimensions les unes après les autres entre crochets.

Code : C++

```
type nomTableau[tailleX][tailleY]
```

Donc pour reproduire le tableau du schéma, on doit déclarer le tableau suivant.

Code : C++ - Déclaration du tableau du schéma

```
int tableau[5][4];
```

Ou encore mieux, en déclarant des constantes.

Code : C++ - Déclaration du tableau du schéma

```
int const tailleX(5);  
int const tailleY(4);  
int tableau[tailleX][tailleY];
```

Et c'est tout ! C'est bien le C++ non ?

Accéder aux éléments

Je suis sûr que je n'ai pas besoin de vous expliquer la suite. Vous avez sûrement deviné tout seul. Pour accéder à une case de notre grille, il faut indiquer la position en X et en Y de la case voulue.

Par exemple `tableau[0][0]` accède à la case en-bas à gauche de la grille. `tableau[0][1]` correspond à celle qui se trouve juste en-dessus, alors que `tableau[1][0]` se situe directement à sa droite.



Comment accéder à la case située en-haut à droite ? 🤖

Hehe. La question piège. C'est la dernière case dans la direction horizontale. Donc entre les premiers crochets, il faut mettre `tailleX-1`, c'est-à-dire 4. C'est également la dernière case selon l'axe vertical et donc il faut spécifier `tailleY-1` entre les seconds crochets. Ce qui donne `tableau[4][3]`.

Aller plus loin

On peut bien sûr aller encore plus loin et créer des grilles tri-dimensionnelles voire même plus. On peut tout à fait déclarer une variable comme ceci :

Code : C++

```
double grilleExtreme[5][4][6][2][7];
```

Mais là, il ne faudra pas me demander un dessin. 😊 Je vous rassure quand même, il est rare de devoir utiliser des grilles à plus de 2 dimensions. Ou alors, c'est que vous prévoyez de faire des programmes vraiment compliqués. 😎

Les strings comme tableaux

Avant de terminer ce chapitre, il faut quand même que je vous fasse une petite révélation. Les chaînes de caractères sont en fait des tableaux! 🤖

On ne le voit pas lors de la déclaration, c'est bien caché. Mais il s'agit en fait d'un tableau de lettres. Il y a même beaucoup de points communs avec les `vector`.

Accéder aux lettres

L'intérêt de voir une chaîne de caractères comme un tableau de lettres, c'est qu'on peut accéder à ces lettres et les modifier. Et je ne vais pas vous surprendre, on utilise aussi les crochets. 😊

Code : C++

```
#include <iostream>  
#include <string>  
using namespace std;
```

```
int main()
{
    string nomUtilisateur("Julien");
    cout << "Vous etes " << nomUtilisateur << "." << endl;

    nomUtilisateur[0] = 'L'; //On modifie la premiere lettre
    nomUtilisateur[2] = 'c'; //On modifie la troisieme lettre

    cout << "Ah non, vous etes " << nomUtilisateur << "!" << endl;

    return 0;
}
```

Testons pour voir.

Code : Console

```
Vous etes Julien.
Ah non, vous etes Lucien!
```

C'est fort ! Mais on peut faire encore mieux.

Les fonctions

On peut également utiliser `size()` pour connaître le nombre de lettres et `push_back()` pour ajouter des lettres à la fin. A nouveau, c'est comme pour `vector`.

Code : C++

```
string texte("Portez ce whisky au vieux juge blond qui fume.");
//46 caractères
cout << "Cette phrase contient " << texte.size() << " lettres." <<
endl;
```

Mais contrairement aux tableaux, on peut ajouter **plusieurs lettres** d'un coup. Et on utilise le `+=`.

Code : C++

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string prenom("Albert");
    string nom("Einstein");

    string total; //Une chaine vide
    total += prenom; //On ajoute le prenom a la chaine vide
    total += " "; //Puis un espace
    total += nom; //Et finalement le nom de famille

    cout << "Vous vous appelez " << total << "." << endl;

    return 0;
}
```

Ce qui donne bien sûr :

Code : Console

```
Vous vous appelez Albert Einstein.
```

C'est fou ce que c'est bien le C++ parfois. 😊

Nous voici donc au terme de ce chapitre. J'espère que vous aimez déjà les tableaux. Vous verrez, ils vont vite devenir des éléments essentiels de vos programmes.

Dans le chapitre suivant, nous allons voir comment lire et écrire des fichiers. Vous serez ensuite livrés à vous-mêmes pour le premier TP de ce cours. 🧑🔧

Lire et écrire des fichiers

Pour l'instant, les programmes que nous avons écrits étaient encore relativement simples. C'est normal, vous débutez. Mais avec un peu d'entraînement, vous seriez capable de créer de vraies applications. Vous commencez à connaître la base du C++. Il vous manque quand même un élément essentiel : l'interaction avec des fichiers.

Jusqu'à maintenant, vous avez appris à écrire dans la console et à récupérer ce que l'utilisateur saisit. Vous serez certainement d'accord avec moi, ce n'est pas suffisant. Pensez à des logiciels comme le bloc-note, votre IDE ou encore un tableur, ce sont tous des programmes qui savent lire et écrire des fichiers. Et même dans le monde des jeux vidéo, on a besoin de ça. Il y a bien sûr les fichiers de sauvegarde, mais aussi les images d'un jeu, les cinématiques, les musiques, etc. En somme, un programme qui ne sait pas interagir avec des fichiers risque d'être très limité.

Voyons donc comment faire ! Vous verrez, si vous maîtrisez l'utilisation de `cin` et de `cout`, alors vous savez déjà presque tout.

Écrire dans un fichier

La première chose à faire quand on veut manipuler des fichiers, c'est de les ouvrir. 😊 Et bien en C++, c'est la même chose.

Une fois le fichier ouvert, tout se passe comme pour `cout` et `cin`. Nous allons, par exemple, retrouver les chevrons `<<` et `>>`. Faites-moi confiance, vous allez rapidement vous y retrouver.

On parle de **flux** pour désigner les moyens de communication d'un programme avec l'extérieur. Dans ce chapitre, nous allons donc parler des **flux vers les fichiers**. Mais dites simplement "lire et écrire des fichiers" quand vous n'êtes pas dans une soirée de programmeurs. 😊

L'en-tête `fstream`

Comme d'habitude en C++, quand on a besoin d'une fonctionnalité, il faut commencer par inclure le bon fichier d'en-tête. Pour les fichiers, il faut spécifier `#include <fstream>` en-haut de notre code source.



Vous connaissez déjà `iostream` qui contient les outils nécessaires aux entrées/sorties vers la console. `iostream` signifie en réalité "input/output stream", ce qui veut dire flux d'entrées/sorties en français. `fstream` correspond à "file stream", flux vers les fichiers en bon français.

La principale différence est qu'il faut un **flux par fichier**. Voyons comment créer un flux sortant, c'est-à-dire un flux permettant d'écrire un fichier.

Ouvrir un fichier en écriture

Les flux sont en réalité des **objets**. Souvenez-vous que le C++ est un langage *orienté objet*. Voici donc un de ces fameux objets.



N'ayez pas peur, il y aura plusieurs chapitres pour en parler. Pour l'instant, prenez-ça comme étant des grosses variables améliorées. Ces objets contiennent beaucoup d'informations sur les fichiers ouverts et proposent quelques fonctionnalités comme fermer le fichier, retourner au début et bien d'autres encore. 😊

L'important pour nous est que l'on déclare un flux exactement de la même manière qu'une variable. Une variable dont le type serait `ofstream` et dont la valeur serait le chemin d'accès du fichier à lire.

Code : C++ - Déclaration d'un flux sortant

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream monFlux("C:/Nanoc/scores.txt");    //Déclaration d'un
    flux permettant d'écrire dans le fichier    //
    C:/Nanoc/scores.txt
    return 0;
}
```

J'ai indiqué le chemin d'accès du fichier entre guillemets. Ce chemin doit être d'une des deux formes suivantes :

- Un chemin absolu. C'est-à-dire montrer l'emplacement du fichier depuis la racine du disque. Par exemple : *C:/Nanoc/C++/Fichiers/scores.txt*.
- Un chemin relatif. C'est-à-dire montrer l'emplacement du fichier depuis l'endroit où se situe le programme sur le disque. Par exemple : *Fichiers/scores.txt* si mon programme se situe dans le dossier *C:/Nanoc/C++/*.

A partir de là, on peut utiliser le flux pour écrire dans le fichier.



Si le fichier n'existait pas, le programme le créerait automatiquement !

Le plus souvent, le nom du fichier est contenu dans une chaîne de caractères `string`. Dans ce cas, il faut utiliser la fonction `c_str()` lors de l'ouverture du fichier.

Code : C++ - Ouverture d'un fichier

```
string const nomFichier("C:/Nanoc/scores.txt");  
  
ofstream monFlux(nomFichier.c_str());    //Déclaration d'un flux  
permettant d'écrire un fichier.
```

Des problèmes peuvent survenir lors de l'ouverture d'un fichier. Si le fichier ne vous appartient pas ou si le disque dur est plein par exemple. C'est pour ça qu'il faut **toujours** tester si tout s'est bien passé. Cela se fait en utilisant la syntaxe `if (monFlux)`. Si ce test n'est pas vrai, alors c'est qu'il y a eu un problème et que l'on ne peut pas utiliser le fichier.

Code : C++ - Tester l'ouverture

```
ofstream monFlux("C:/Nanoc/scores.txt"); //On essaye d'ouvrir le  
fichier  
  
if(monFlux)    //On teste si tout est OK.  
{  
    //Tout est OK. On peut utiliser le fichier  
}  
else  
{  
    cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;  
}
```

Tout est donc prêt pour l'écriture. Et vous allez voir que ce n'est pas vraiment nouveau. 😊

Écrire dans un flux

Je vous avais dit que tout était comme pour `cout`. C'est donc sans surprise que je vous présente le moyen d'envoyer des informations dans un flux. C'est les chevrons (`<<`) qu'il faut utiliser.

Code : C++ - Écriture dans un fichier

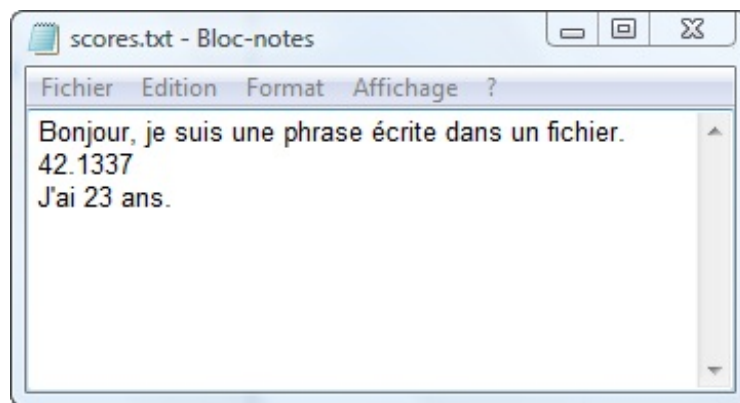
```
#include <iostream>  
#include <fstream>
```

```
#include <string>
using namespace std;

int main()
{
    string const nomFichier("C:/Nanoc/scores.txt");
    ofstream monFlux(nomFichier.c_str());

    if(monFlux)
    {
        monFlux << "Bonjour, je suis une phrase écrite dans un
fichier." << endl;
        monFlux << 42.1337 << endl;
        int age(23);
        monFlux << "J'ai " << age << " ans." << endl;
    }
    else
    {
        cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
    }
    return 0;
}
```

Si j'exécute ce programme, je retrouve ensuite sur mon disque un fichier scores.txt dont voici le contenu :



Essayez par vous-mêmes !

Vous pouvez par exemple écrire un programme qui demande son nom et son âge à l'utilisateur et qui écrit ces données dans un fichier.

Les différents modes d'ouverture

Il ne nous reste plus qu'un petit point à régler.



Que se passe-t-il si le fichier existe déjà ?

Il sera supprimé et remplacé par ce que vous écrivez, ce qui n'est pas bien si l'on souhaite ajouter des informations à la fin d'un fichier pré-existant. Pensez par exemple à un fichier qui contiendrait la liste des actions effectuées par l'utilisateur. On ne veut pas tout effacer à chaque fois. On veut juste y ajouter des lignes.

Pour pouvoir écrire à la fin d'un fichier, il faut le spécifier lors de l'ouverture en ajoutant un deuxième paramètre lors de la création du flux :

```
ofstream monFlux("C:/Nanoc/scores.txt", ios::app);
```



app est un raccourci pour *append*, le verbe anglais qui signifie "ajouter à la fin".



Avec ça, plus de problème d'écrasement des données. Tout ce qui sera écrit sera ajouté à la fin.

Lire un fichier

Nous avons appris à écrire dans un fichier, voyons maintenant comment fonctionne la lecture de fichier. Vous allez voir, ce n'est pas très différent de ce que vous connaissez déjà.

Ouvrir un fichier en lecture...

Le principe est exactement le même. On va simplement utiliser un `ifstream` au lieu d'un `ofstream` et il faut également tester l'ouverture. C'est bien le C++ quand même. 😊

Code : C++ - Lecture d'un fichier

```
ifstream monFlux("C:/Nanoc/C++/data.txt"); //Ouverture d'un fichier
en lecture

if(monFlux)
{
    //Tout est prêt pour la lecture.
}
else
{
    cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." <<
endl;
}
```

Rien de bien nouveau. 😊

... et le lire

Il y a trois manières différentes de lire un fichier.

1. Ligne par ligne en utilisant `getline()`.
2. Mot par mot en utilisant les chevrons `>>`.
3. Caractère par caractère en utilisant `get()`.

Voyons ces trois moyens en détail.

Lire ligne par ligne

La première méthode permet de récupérer une ligne entière et de la stocker dans une chaîne de caractères.

Code : C++

```
string ligne;
getline(monFlux, ligne); //On lit une ligne complète
```

Le fonctionnement est exactement le même qu'avec `cin`. Vous savez donc déjà tout. 😊

Lire mot par mot

La deuxième manière de faire, vous la connaissez aussi. Comme je suis gentil, je vous propose quand même un petit rappel.

Code : C++

```
double nombre;  
monFlux >> nombre; //Lit un nombre à virgule depuis le fichier  
string mot;  
monFlux >> mot; //Lit un mot depuis le fichier
```

Cette méthode lit ce qui se trouve entre l'endroit où l'on se situe dans le fichier et le prochain espace. Ce qui est lu est alors traduit en **double**, **int** ou **string** selon le type de variable dans lequel on écrit.

Lire caractère par caractère

Finalement, il nous reste la dernière méthode. La seule réellement nouvelle. Mais tout aussi simple, je vous rassure.

Code : C++ - Lecture d'un caractère

```
char a;  
monFlux.get(a);
```

Ce code lit *une seule* lettre et la stocke dans la variable **a**.



Cette méthode lit réellement **tous** les caractères. Les espaces, retours à la ligne et tabulations sont, entre autres, lus par cette fonction. Bien que bizarres, ces caractères seront néanmoins stockés dans la variable.

Lire un fichier en entier

On veut très souvent lire un fichier en entier. Je vous ai montré comment lire, mais pas comment s'arrêter quand on arrive à la fin !

Pour savoir si l'on peut continuer à lire, il faut utiliser la valeur retournée par la fonction `getline()`. En effet, en plus de lire une ligne, cette fonction renvoie un **bool** indiquant si l'on peut continuer à lire. Si la fonction renvoie **true**, tout va bien, la lecture peut continuer. Si elle renvoie **false**, c'est qu'on est arrivé à la fin du fichier ou qu'il y a eu une erreur. Dans les deux cas, il faut s'arrêter de lire.

Vous vous rappelez des boucles ? On cherche à lire le fichier *tant qu'on n'a pas atteint la fin*. La boucle **while** est donc le meilleur choix. Voici comment faire :

Code : C++ - Lecture d'un fichier du début à la fin

```
#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main()  
{  
    ifstream fichier("C:/Nanoc/fichier.txt");  
  
    if(fichier)  
    {  
        //L'ouverture s'est bien passée. On peut donc lire  
  
        string ligne;        //Une variable pour stocker les lignes  
  
        while (fichier.getline(ligne, 1000))  
        {  
            //Affichage de la ligne  
            cout << ligne << endl;  
        }  
    }  
}
```

```

        while(getline(fichier, ligne))    //Tant qu'on n'est pas a
la fin, on lit
        {

            cout << ligne << endl; //Et on l'affiche dans la
console
                                //ou alors on fait quelque chose
avec cette ligne
                                //A vous de voir :)

        }
        else
        {
            cout << "ERREUR: Impossible d'ouvrir le fichier en lecture."
<< endl;
        }

        return 0;
    }

```

Une fois que l'on a lu les lignes, on peut les manipuler facilement. Ici, j'affiche simplement les lignes, mais dans un programme réel on les utiliserait autrement. La seule limite est votre imagination. 😊

C'est la méthode la plus utilisée pour lire un fichier. Une fois que l'on a récupéré les lignes dans une variable `string`, on peut facilement travailler dessus grâce aux fonctions utilisables sur les chaînes de caractères.

Quelques astuces

Il ne reste que quelques astuces à voir et vous saurez alors tout ce qu'il faut sur les fichiers. 😊

Fermer prématurément un fichier

Je vous ai expliqué en tout début de chapitre comment ouvrir un fichier. Mais je ne vous ai pas montré comment le refermer. Ce n'est pas un oubli de ma part, il s'avère juste que ce n'est pas nécessaire. Les fichiers ouverts sont automatiquement refermés lorsque l'on sort du bloc où le flux est déclaré.

Code : C++

```

void f()
{
    ofstream flux("C:/Nanoc/data.txt");    //Le fichier est ouvert

    //Utilisation du fichier

}    //Lorsque l'on sort du bloc, le fichier est automatiquement
refermé

```

Il n'y a donc rien à faire. Aucun risque d'oublier de refermer le fichier ouvert. 😊

Il arrive par contre qu'on ait besoin de fermer le fichier avant sa "fermeture automatique". Il faut alors utiliser la fonction `close()` des flux.

Code : C++

```

void f()
{
    ofstream flux("C:/Nanoc/data.txt");    //Le fichier est ouvert

    //Utilisation du fichier

    flux.close(); //On referme le fichier

```

```
d'ici          //On ne peut plus écrire dans le fichier à partir
}
```

De la même manière, il est possible de retarder l'ouverture d'un fichier après la déclaration du flux en utilisant la fonction `open()`.

Code : C++

```
void f()
{
    ofstream flux;    //Un flux sans fichier associé

    flux.open("C:/Nanoc/data.txt"); //On ouvre le fichier
    C:/Nanoc/data.txt

    //Utilisation du fichier

    flux.close(); //On referme le fichier
                //On ne peut plus écrire dans le fichier à partir
d'ici
}
```

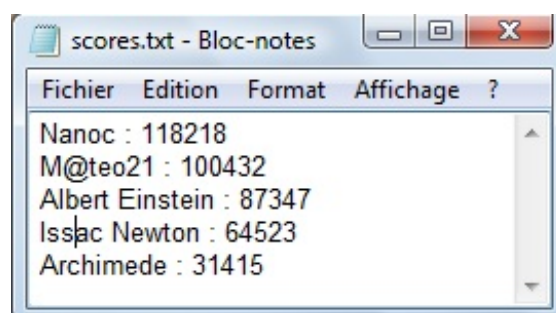
Comme vous le voyez, c'est très simple. Mais dans la majorité des cas, c'est inutile. 😊 Ouvrir directement le fichier et le laisser se fermer automatiquement est suffisant.



Certaines personnes aiment utiliser `open()` et `close()`, alors que ce n'est pas nécessaire. On peut ainsi mieux voir où le fichier est ouvert et où il est refermé. C'est une question de goût. A vous de voir ce que vous préférez.

Le curseur dans le fichier

Plongeons un petit peu plus dans les détails techniques. Voyons comment se déroule la lecture. Quand on ouvre un fichier dans le bloc-note, par exemple, il y a un curseur qui indique l'endroit où l'on va écrire. Sur l'image suivante, le curseur se situe après les deux "s" sur la 4^e ligne.



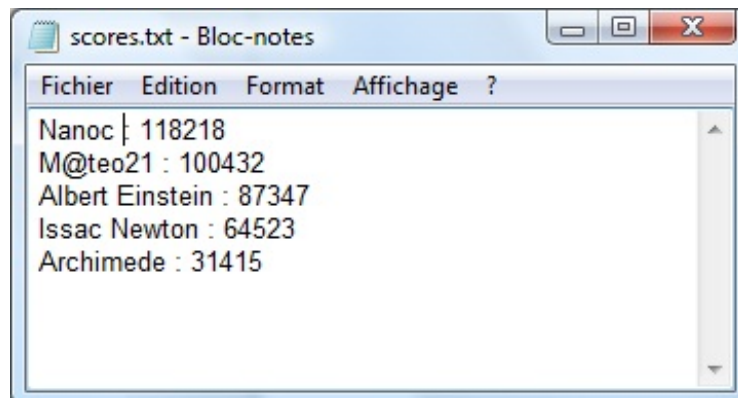
Si l'on tape sur une touche du clavier, une lettre sera ajoutée à cet endroit du fichier. J'imagine que je ne vous apprend rien en disant ça. 😊 Ce qui est plus intéressant, c'est qu'en C++ il y a aussi en quelque sorte un curseur.

Lorsque l'on écrit la ligne suivante :

Code : C++ - Curseur au début du fichier

```
ifstream fichier("C:/Nanoc/scores.txt")
```

le fichier `C:/Nanoc/scores.txt` est ouvert et le curseur est placé tout au début du fichier. Si on lit le premier mot du fichier, on obtient bien sûr la chaîne de caractères `"Nanoc"` puisque c'est le premier mot du fichier. En faisant ça, le "curseur C++" se déplace jusqu'au début du mot suivant. Comme sur l'image suivante :



Le mot suivant qui pourra être lu sera donc `": "`, puis `118218` et ainsi de suite jusqu'à la fin. On est donc obligés de lire un fichier **dans l'ordre**. Ce n'est pas très pratique. 😞

Heureusement, il existe des moyens de se déplacer dans un fichier. On peut par exemple dire "je veux placer le curseur 20 caractères après le début" ou "je veux avancer le curseur de 32 caractères". On peut ainsi lire que les parties qui nous intéressent réellement.

La première chose à faire est de savoir où se situe le curseur. Dans un deuxième temps, on pourra le déplacer. Voici comment.

Connaître sa position

Il existe une fonction permettant de savoir à quel octet du fichier on se trouve. Autrement dit, elle permet de savoir à quel caractère du fichier on se situe. Malheureusement, cette fonction n'a pas le même nom pour les flux entrants et sortants. Et en plus ce sont des noms bizarres. 😞 Je vous ai mis les noms des deux fonctions dans un petit tableau

Pour ifstream	Pour ofstream
<code>tellg()</code>	<code>tellp()</code>

Par contre, elles s'utilisent toutes les deux de la même manière.

Code : C++

```
ofstream fichier("C:/Nanoc/data.txt");

int position = fichier.tellp(); //On récupère la position

cout << "Nous nous situons au " << position << "eme caractere du
fichier." << endl;
```

Se déplacer

A nouveau, il existe deux fonctions. Une pour chaque type de flux.

Pour ifstream	Pour ofstream
<code>seekg()</code>	<code>seekp()</code>

Elles s'utilisent à nouveau de la même manière, je ne vous présente donc qu'une des deux versions. 😎

Ces fonctions reçoivent deux arguments. Une position dans le fichier et un nombre de caractères **à ajouter** à cette position :

Code : C++

```
flux.seekp(nombreCaracteres, position);
```

Les trois positions possibles sont :

- Le début du fichier : `ios::beg`.
- La fin du fichier : `ios::end`.
- La position actuelle : `ios::cur`.

Si par exemple, je souhaite me placer 10 caractères après le début du fichier, j'utilise `flux.seekp(10, ios::beg)` ;. Si je souhaite aller 20 caractères plus loin que l'endroit où le curseur se situe, j'utilise `flux.seekp(20, ios::cur)` ;. Je pense que vous avez compris. 😊

Voilà donc notre problème de lecture résolu.

Connaître la taille d'un fichier

Cette troisième astuce utilise en réalité les deux précédentes. Pour connaître la taille d'un fichier, on se déplace à la fin et on demande au flux de nous dire où il se trouve. Vous voyez comment faire ? 🤔

Bon, je vous montre.

Code : C++ - Taille d'un fichier

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream fichier("C:/Nanoc/meilleursScores.txt"); //On ouvre le
    fichier
    fichier.seekg(0, ios::end); //On se déplace a la fin du fichier

    int taille;
    taille = fichier.tellg(); //On récupère la position qui
    correspond donc a la taille du fichier !

    cout << "Taille du fichier : " << taille << " octets." << endl;

    return 0;
}
```

Je suis sûr que vous le saviez !

Voilà, on a fait le tour des notions principales. Vous êtes prêts à vous lancer seuls dans le vaste monde des fichiers. Avec ces nouvelles notions, vous êtes prêts pour entrer dans la cour des grands et réaliser de vrais programmes. Et ça tombe bien, puisque le prochain chapitre sera un TP dans lequel vous aurez besoin de tout ce que vous avez appris précédemment !

Je vous conseille de bien réviser les parties qui vous semblaient compliquées ! 🧐

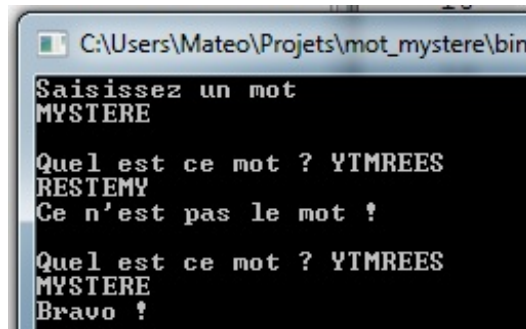
[TP] Le mot mystère

Depuis le début de ce cours sur le C++, vous avez découvert de nombreuses notions : le compilateur, l'IDE, les variables, les fonctions, les conditions, les boucles... Vous avez pu voir des exemples d'utilisation de ces notions au fur et à mesure, mais est-ce que vous avez pris le temps de créer un **vrai programme** pour vous entraîner ? Non ? Eh bien c'est le moment de s'y mettre !

On trouve régulièrement des TP au milieu des cours du Site du Zéro. Celui-ci ne fait pas exception.

Le but ? Vous forcer à vous lancer "pour de vrai" dans la programmation. Parce que je sais bien qu'il y en a beaucoup parmi vous qui ont à peine ouvert leur IDE pour le moment, ce TP est l'occasion de vous y mettre vraiment. 😊

Le sujet de ce TP n'est pas *très* compliqué mais promet d'être amusant : nous allons mélanger les lettres d'un mot et demander à un joueur de retrouver le mot "mystère" qui se cache derrière ces lettres. 😊



Préparatifs et conseils

Le jeu que nous voulons réaliser consiste à retrouver un mot dont les lettres ont été mélangées. C'est simple en apparence, mais il va nous falloir utiliser des notions que nous avons découvertes dans les chapitres précédents :

- Les variables string
- Les fonctions
- Les structures de contrôle (boucles, conditions...)

N'hésitez pas à relire rapidement ces chapitres pour bien être dans le bain avant de commencer ce TP !

Principe du jeu "Le mot mystère"

Nous voulons réaliser un jeu qui se déroule de la façon suivante :

1. Le joueur 1 saisit un mot au clavier
2. L'ordinateur mélange les lettres du mot
3. Le joueur 2 essaie de deviner le mot d'origine à partir des lettres mélangées

Voici un exemple de partie du jeu que nous allons réaliser :

Code : Console

```
Saisissez un mot
MYSTERE

Quel est ce mot ? MSERETY
RESEMTY
Ce n'est pas le mot !

Quel est ce mot ? MSERETY
MYRESTE
Ce n'est pas le mot !
```



```
Quel est ce mot ? MSERETY  
MYSTERE  
Bravo !
```

1. Dans cette partie, le joueur 1 choisit "MYSTERE" comme mot à deviner.
2. L'ordinateur mélange les lettres et demande au joueur 2 de retrouver le mot qui se cache derrière "MSERETY".
3. Le joueur 2 essaie de trouver le mot. Ici, il y parvient au bout de 3 essais :
 1. RESEMTY : on lui dit que ce n'est pas ça
 2. MYRESTE : là non plus
 3. MYSTERE : là on lui dit bravo car il a trouvé, et le programme s'arrête. 😊

Bien sûr, le joueur 2 peut actuellement facilement lire le mot saisi par le joueur 1. Nous verrons à la fin du TP comment nous pouvons améliorer ça.

Quelques conseils pour bien démarrer

Quand on lâche un débutant dans la nature la première fois, avec comme seule instruction "Allez, code-moi ça", il est en général assez désarmé.

"Par quoi dois-je commencer ?", "Qu'est-ce que je dois faire, qu'est-ce que je dois utiliser ?". Bref, il ne sait pas du tout s'y prendre, et c'est bien normal vu qu'il n'a jamais fait ça. 😊

Mais moi, je n'ai pas envie que vous vous perdiez ! Je vais donc vous donner une série de conseils pour que vous soyez le mieux préparés possible. Bien entendu, ce sont juste des conseils, vous en faites ce que vous voulez. 😊

Repérez les étapes du programme

Je vous ai décrit les 3 étapes du programme un peu plus tôt :

1. Saisie du mot à deviner
2. Mélange des lettres
3. Boucle qui se répète tant que le mot mystère n'a pas été trouvé

Ces étapes sont en fait assez indépendantes. Plutôt que d'essayer de réaliser tout le programme d'un coup, pourquoi vous n'essayez pas de faire chaque étape indépendamment des autres ?

1. L'étape 1 est très simple : l'utilisateur doit saisir un mot qu'on va stocker en mémoire (dans une string, car c'est le type adapté). Si vous connaissez cout et cin, vous ne mettrez pas plus de quelques minutes à écrire le code correspondant.
2. L'étape 2 est la plus complexe : vous avez une string qui contient un mot comme MYSTERE et vous voulez aléatoirement mélanger les lettres pour obtenir quelque chose comme MSERETY. Comment faire ? Je vais vous aider un peu pour ça car vous devez utiliser certaines choses que nous n'avons pas vues.
3. L'étape 3 est de difficulté moyenne : vous devez créer une boucle qui demande de saisir un mot et qui le compare au mot mystère. La boucle s'arrête dès que le mot saisi est identique au mot mystère.

Créez un canevas de code avec les étapes

Comme vous le savez, tous les programmes contiennent une fonction main(). Ecrivez dès maintenant des commentaires pour séparer les principales étapes du programme. Ça devrait donner quelque chose comme ça :

Code : C++

```
int main()  
{  
    // 1 : On demande à saisir un mot
```

```
// 2 : On mélange les lettres du mot  
  
// 3 : On demande à l'utilisateur quel est le mot mystère  
  
return 0;  
}
```

A vous de réaliser les étapes ! Pour y aller en difficulté croissante, je vous conseille de faire d'abord l'étape 1, puis l'étape 3 et enfin l'étape 2.

Lorsque vous aurez réalisé les étapes 1 et 3, le programme vous demandera un mot et vous devrez le ressaisir. Ce ne sera pas très amusant mais comme ça vous pourrez valider que vous avez réussi les premières étapes ! N'hésitez donc pas à y aller pas à pas !

Ci-dessous un aperçu du programme "intermédiaire" avec seulement les étapes 1 et 3 réalisées :

Code : Console

```
Saisissez un mot  
MYSTERE  
  
Quel est ce mot ?  
RESEMTY  
Ce n'est pas le mot !  
  
Quel est ce mot ?  
MYRESTE  
Ce n'est pas le mot !  
  
Quel est ce mot ?  
MYSTERE  
Bravo !
```

Comme vous le voyez, le programme ne propose pas encore le mot avec les lettres mélangées, mais si vous arrivez déjà à faire ça vous avez fait 50% du travail ! 😊

Un peu d'aide pour mélanger les lettres

L'étape de mélange des lettres est la plus "difficile" (si je puis dire !) de ce TP. Je vous donne quelques informations et conseils pour réaliser cette fameuse étape n°2.

Tirer un nombre au hasard

Pour que les lettres soient aléatoirement mélangées, vous allez devoir tirer un nombre au hasard. Nous n'avons pas appris à le faire auparavant, il faut donc que je vous explique comment ça fonctionne.

- Vous devez inclure `ctime` et `cstdlib` au début de votre code source pour obtenir les fonctionnalités de nombres aléatoires.
- Vous devez appeler la fonction `srand(time(0))` ; une seule fois au début de votre programme (au début du `main`) pour initialiser la génération des nombres aléatoires.
- Et enfin, pour générer un nombre compris entre 0 et 4 (par exemple), vous écrirez : `nbAleatoire = rand() % 5;` (on écrit 5 pour avoir un nombre entre 0 et 4, oui oui 😊).

Un exemple qui génère un nombre entre 0 et 4 :

Code : C++

```
#include <iostream>
```

```
#include <ctime> // Obligatoire
#include <cstdlib> // Obligatoire

using namespace std;

int main()
{
    int nbAleatoire(0);

    srand(time(0));

    nbAleatoire = rand() % 5;

    return 0;
}
```

Tirer une lettre au hasard



Tirer un nombre au hasard c'est bien, mais pour ce programme j'ai besoin de tirer une lettre au hasard pour mélanger les lettres !

Imaginons que vous ayez une string appelée motMystere qui contient le mot mystère. Vous avez appris que les string pouvaient être considérées comme des tableaux, souvenez-vous ! Ainsi, motMystere[0] correspond à la première lettre, motMystere[1] à la deuxième lettre, etc.

Il suffit de générer un nombre aléatoire entre 0 et le nombre de lettres du mot (qui nous est donné par motMystere.size()) pour tirer une lettre au hasard ! Une petite idée de code pour récupérer une lettre au hasard :

Code : C++

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    string motMystere("MYSTERE");

    srand(time(0));

    int position = rand() % motMystere.size();

    cout << "Lettre au hasard :" << motMystere[position];

    return 0;
}
```

Retirer une lettre d'une string

Pour éviter de tirer 2 fois la même lettre d'un mot, je vous conseille de retirer au fur et à mesure les lettres qui ont été piochées. Pour ce faire, on va faire appel à erase() sur le mot mystère comme ceci :

Code : C++

```
motMystere.erase(4, 1); // Retire la lettre n°5
```

Il y a 2 paramètres :

- Le numéro de la lettre à retirer du mot (ici 4, ce qui correspond à la 5ème lettre car on commence à compter à partir de 0).
- Le nombre de lettres à retirer (ici 1).

Créez des fonctions !

Ce n'est pas une obligation, mais plutôt que de tout mettre dans le `main()`, vous pourriez créer des fonctions qui ont des rôles spécifiques. Par exemple, l'étape 2 qui génère un mot dont les lettres ont été mélangées mériterait d'être faite dans une fonction.

Ainsi, on pourrait appeler la fonction comme ceci dans le `main()` :

Code : C++

```
motMelange = melangerLettres(motMystere);
```

On lui envoie le `motMystere`, elle nous retourne un `motMelange`. 😊

Bien entendu, toute la difficulté consiste ensuite à coder cette fonction `melangerLettres`. Allez au boulot ! 😊

Correction

C'est l'heure de la correction !

Vous avez sûrement passé du temps à réfléchir à ce programme, ça n'a peut-être pas toujours été facile et vous n'avez pas forcément su tout faire. Ce n'est pas grave ! Ce qui compte, c'est d'avoir essayé : c'est comme ça que vous progressez le plus !

Normalement, les étapes 1 et 3 étaient assez faciles pour tout le monde. Seule l'étape 2 (mélange des lettres) demandait plus de réflexion : je l'ai isolée dans une fonction `melangerLettres` comme je vous l'ai suggéré plus tôt.

Le code

Sans plus attendre, voici la correction :

Code : C++

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>

using namespace std;

string melangerLettres(string mot)
{
    string melange;
    int position(0);

    // Tant que nous n'avons pas extrait toutes les lettres du mot
    while (mot.size() != 0)
    {
        // On choisit un numéro de lettre au hasard dans le mot
        position = rand() % mot.size();
        // On ajoute la lettre dans le mot mélangé
        melange += mot[position];
    }
}
```

```

        melange += mot[position],
        // On retire cette lettre du mot mystère pour ne pas la
        prendre une 2e fois
        mot.erase(position, 1);
    }

    // On renvoie le mot mélangé
    return melange;
}

int main()
{
    string motMystere, motMelange, motUtilisateur;

    // Initialisation des nombres aléatoires
    srand(time(0));

    // 1 : On demande à saisir un mot
    cout << "Saisissez un mot" << endl;
    cin >> motMystere;

    // 2 : On récupère le mot avec les lettres mélangées dans
    motMelange
    motMelange = melangerLettres(motMystere);

    // 3 : On demande à l'utilisateur quel est le mot mystère
    do
    {
        cout << endl << "Quel est ce mot ? " << motMelange << endl;
        cin >> motUtilisateur;

        if (motUtilisateur == motMystere)
        {
            cout << "Bravo !" << endl;
        }
        else
        {
            cout << "Ce n'est pas le mot !" << endl;
        }
    } while (motUtilisateur != motMystere); // On recommence tant
    qu'il n'a pas trouvé

    return 0;
}

```

Ne vous laissez pas surprendre par la "taille" du code (qui n'est d'ailleurs pas très gros) et soyez méthodiques en le lisant : commencez par lire le `main()` et non la fonction `melangerLettres()`. Regardez les différentes étapes du programme une par une : isolées, elles sont plus simples à comprendre.

Des explications

Voici quelques explications pour mieux comprendre le programme, étape par étape.

Etape 1 : saisir un mot

C'était, de loin, l'étape la plus simple : un `cout` pour afficher un message, un `cin` pour récupérer un mot que l'on stocke dans la variable `motMystere`. Facile !

Etape 2 : mélanger les lettres

Plus difficile, cette étape est réalisée en fait dans une fonction `melangerLettres` (en haut du programme). Le `main()` appelle la fonction `melangerLettres()` en lui envoyant le mot mystère. Le but de la fonction est de retourner une version mélangée des lettres, que l'on stocke dans `motMelange`.

Analysons la fonction `melangerLettres`. Elle extrait une à une les lettres du mot aléatoirement et recommence tant qu'il reste des lettres à extraire dans le mot :

Code : C++

```
while (mot.size() != 0)
{
    position = rand() % mot.size();
    melange += mot[position];
    mot.erase(position, 1);
}
```

A chaque passage de boucle, on tire un nombre au hasard compris entre 0 et le nombre de lettres qu'il reste dans le mot. On ajoute ces lettres piochées aléatoirement dans une string `melange` et on retire les lettres du mot d'origine pour ne pas les piocher une deuxième fois.

Une fois toutes les lettres extraites, on sort de la boucle et on retourne la variable `melange` qui contient les lettres dans le désordre.

Etape 3 : demander à l'utilisateur le mot mystère

Cette étape prend la forme d'une boucle `do ... while`, qui nous permet de s'assurer qu'on demande bien au moins une fois quel est le mot mystère.

L'utilisateur saisit un mot grâce à `cin`, et on compare ce mot avec le `motMystere` qu'il faut trouver. On continue la boucle tant que le mot n'a pas été trouvé, d'où la condition :

Code : C++

```
}while (motUtilisateur != motMystere); // On recommence tant qu'il
n'a pas trouvé
```

On affiche un message différent selon si on a trouvé ou non le mot mystère. Le programme s'arrête dès qu'on est sorti de la boucle, donc dès qu'on a trouvé le mot mystère. 😊

Téléchargement

Vous pouvez télécharger le programme avec le lien suivant :

[Télécharger le code source du jeu "Mot mystère"](#)

Le fichier ZIP contient :

- `main.cpp` : le fichier source du programme (l'essentiel !)
- `mot_mystere.cbp` : le fichier de projet Code::Blocks (facultatif, pour ceux qui utilisent cet IDE)

Vous pouvez ainsi tester le programme et éventuellement vous en servir comme base par la suite pour réaliser les améliorations que je vais vous proposer (si vous n'avez pas réussi à faire le programme vous-même bien entendu !).

Aller plus loin

Notre programme est terminé... mais on peut toujours l'améliorer. Je vais vous présenter une série de suggestions pour aller plus loin, ce qui vous donnera l'occasion de travailler un peu plus sur ce petit jeu. 😊

Ces propositions sont de difficulté croissante :

- **Ajoutez des sauts de ligne au début** : lorsque le premier joueur saisit le mot mystère la première fois, vous devriez créer plusieurs sauts de ligne pour que le joueur 2 ne voie pas le mot qui a été saisi, sinon c'est trop facile pour lui. 😊
Utilisez plusieurs `endl` par exemple pour créer plusieurs retours à la ligne.
- Proposez au joueur de faire une **nouvelle partie**. Actuellement, une fois le mot trouvé, le programme s'arrête. Et si vous demandiez *"Voulez-vous faire une autre partie ? (o/n)"*. En fonction de la réponse saisie, vous reprenez au début du programme. Pour ce faire, il faudra créer une grosse boucle `do...while` qui englobe les 3 étapes du programme.
- **Fixez un nombre maximal de coups** pour trouver le mot mystère. Vous pouvez par exemple indiquer "Il vous reste 5 essais" et lorsque les 5 essais sont écoulés, le programme s'arrête en affichant la solution.
- Calculez le **score moyen du joueur** à la fin du programme : après plusieurs parties du joueur, affichez-lui son score. Ce score sera la moyenne des parties précédentes. Vous pouvez calculer le nombre de points comme vous le voulez. Vous devrez sûrement utiliser les tableaux dynamiques vector pour stocker les scores de chaque partie au fur et à mesure avant d'en faire la moyenne.
- **Piochez le mot dans un fichier-dictionnaire** : pour que l'on puisse jouer seul, vous pourriez créer un fichier contenant une série de mots (un par ligne) dans lequel le programme va aller piocher aléatoirement à chaque fois. Voici un exemple de fichier-dictionnaire :

Code : Autre

```
MYSTERE
XYLOPHONE
ABEILLE
PLUTON
MAGIQUE
AVERTISSEMENT
```

Au lieu de demander le mot à deviner (étape 1) on va chercher dans un fichier comme celui-ci un mot aléatoirement. A vous d'utiliser les fonctionnalités de lecture de fichiers que vous avez apprises ! 😊

... Allez, puisque vous m'êtes sympathiques, je vous propose même de télécharger un fichier-dictionnaire tout prêt avec des dizaines de milliers de mots ! Merci qui ?! 😊

[Télécharger le fichier-dictionnaire \(600 Ko\)](#)

Si vous avez d'autres idées, n'hésitez pas à compléter encore ce programme ! Cela vous fera beaucoup progresser, vous verrez. 😊

Je suis sûr que ce premier TP vous aura fait bien plus progresser que tous les chapitres précédents réunis. 😊

Ne vous arrêtez pas au seul sujet du TP : essayez de faire les améliorations proposées, trouvez d'autres améliorations à faire, et lancez-vous ! Si vous avez un problème et que vous avez besoin d'aide, n'oubliez pas que [le forum C++](#) est à votre disposition. 😊

Les pointeurs

Nous voilà dans le dernier chapitre de présentation des bases du C++. Accrochez-vous car le niveau monte d'un cran ! Le sujet des pointeurs fait peur à beaucoup de monde et c'est certainement un des chapitres les plus complexes de ce cours. Une fois cet écueil passé, beaucoup de choses vous paraîtront plus simples et plus claires.

Les pointeurs sont utilisés dans **tous** les programmes C++, même si vous n'en avez pas eu conscience jusque là. Il y en a partout. Pour l'instant, ils étaient cachés et vous n'avez pas eu à en manipuler directement. Cela va changer avec ce chapitre. Nous allons apprendre à gérer très finement ce qui se passe dans la mémoire de l'ordinateur.

C'est un chapitre plutôt difficile, il est donc normal que vous ne compreniez pas tout du premier coup. N'ayez pas peur de le relire une seconde fois dans quelques jours pour vous assurer que vous avez bien tout assimilé !

Une question d'adresse

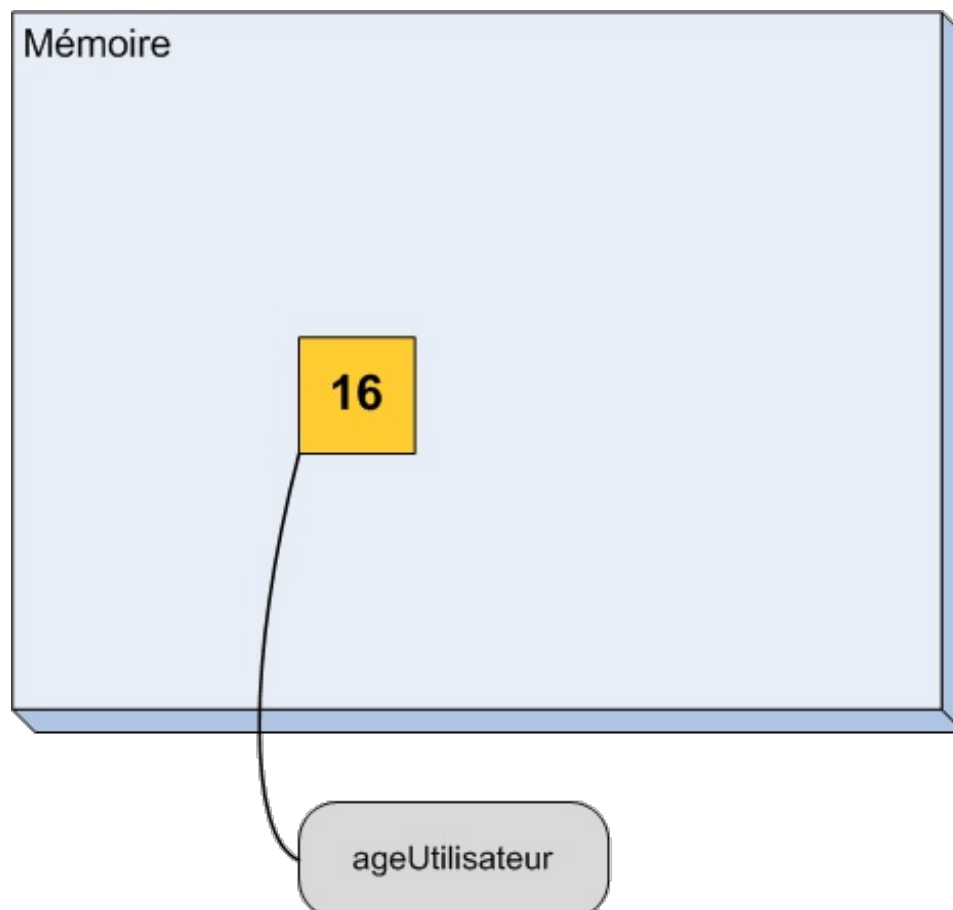
Est-ce que vous vous rappelez du [chapitre sur la mémoire](#) ? Oui, oui, celui qui présentait la notion de variable. Je vous invite à le relire et surtout à vous remémorer les différents schémas.

Je vous avais dit que lorsque l'on déclare une variable, l'ordinateur nous prête une place dans sa mémoire et y accroche une étiquette portant le nom de la variable.

Code : C++

```
int main()
{
    int ageUtilisateur(16);
    return 0;
}
```

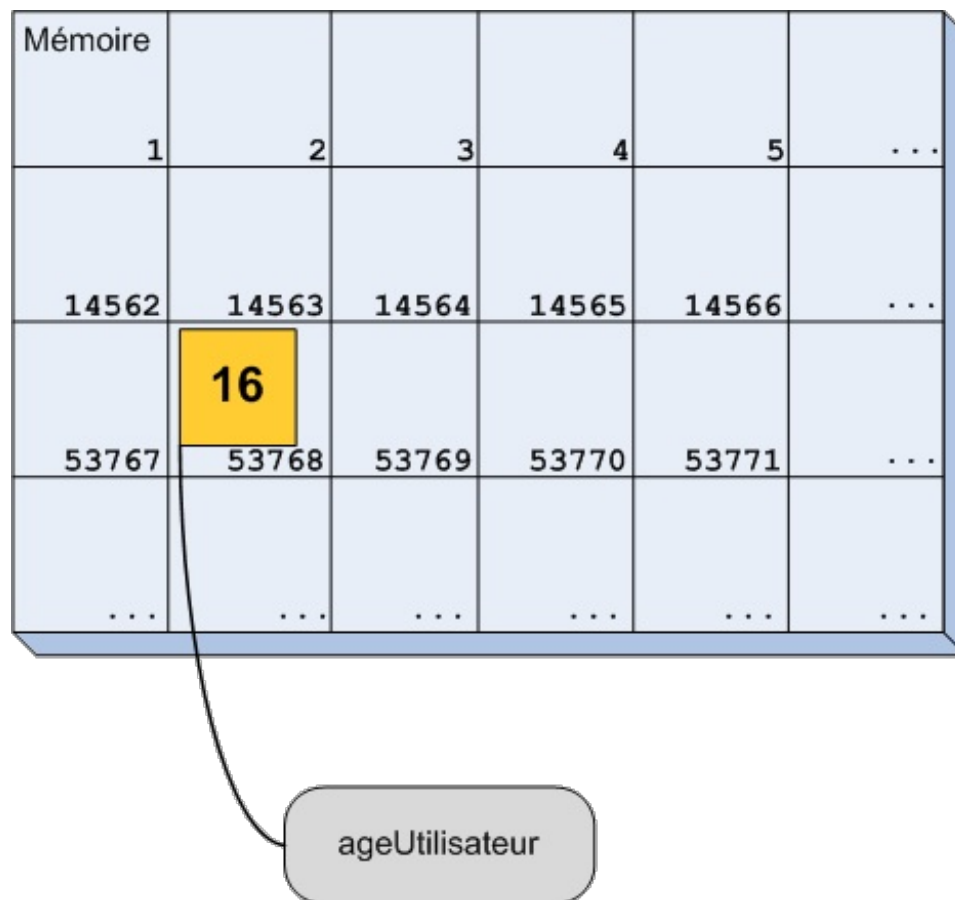
On pouvait donc représenter la mémoire utilisée dans ce programme sur le schéma suivant :



C'était simple et beau. Malheureusement, je vous ai un peu menti. Je vous ai simplifié les choses !

Vous commencez à le savoir, dans un ordinateur tout est bien ordonné et rangé de manière logique. Le système des étiquettes dont je vous ai parlé n'est donc pas tout à fait correct.

La mémoire d'un ordinateur est réellement constituée de "cases", là je ne vous ai pas menti. Il y en a même énormément. Plusieurs milliards sur un ordinateur récent ! Il faut donc un système pour que l'ordinateur puisse retrouver les cases dont il a besoin. Chaque "case" possède un numéro unique, son **adresse**.



Sur le schéma, on voit cette fois toutes les cases de la mémoire avec leur adresse. Notre programme utilise une seule de ces cases, la **53768**, pour y stocker sa variable.



On ne peut PAS mettre deux variables dans la même case.

L'important est que chaque variable possède une seule adresse. Et chaque adresse correspond à une seule variable.

L'adresse est donc un deuxième moyen d'accéder à une variable. On peut accéder à la case jaune du schéma par deux chemins différents :

- On peut passer par **son nom** (l'étiquette) comme on sait déjà le faire...
- Mais on peut aussi accéder à la variable grâce à son **adresse** (son numéro de case).. On pourrait alors dire à l'ordinateur "Affiche moi le contenu de l'adresse 53768" ou encore "Additionne les contenus des adresses 1267 et 91238".

Est-ce que ça vous tente d'essayer ? 🤖 Vous vous demandez peut-être à quoi ça peut bien servir. Utiliser l'étiquette était un moyen simple et efficace. C'est vrai. Mais nous verrons plus loin que passer par les adresses peut parfois être nécessaire.

Commençons par voir comment connaître l'adresse d'une variable.

Afficher l'adresse

En C++, le symbole pour obtenir l'adresse d'une variable est l'esperluette (&). Si je veux afficher l'adresse de la variable `ageUtilisateur`, je dois donc écrire `&ageUtilisateur`. Essayons.

Code : C++

```
#include <iostream>
using namespace std;

int main()
{
    int ageUtilisateur(16);
    cout << "L'adresse est : " << &ageUtilisateur << endl; //
    Affichage de l'adresse de la variable
    return 0;
}
```

Chez moi, j'obtiens le résultat suivant :

Code : Console

```
L'adresse est : 0x22ff1c
```



Vous aurez certainement un résultat différent. La case peut changer d'une exécution à l'autre du programme. 😊

Même si elle contient des lettres, cette adresse est un nombre. Celui-ci est juste écrit en hexadécimal (en base 16), une autre façon d'écrire les nombres. Les ordinateurs aiment bien travailler dans cette base. Pour information, en base 10, dans notre écriture courante, cette adresse correspond à **2 293 532**. Ce n'est pas une information très intéressante cependant. 😊

Ce qui est sûr c'est qu'afficher une adresse est très rarement utile. Souvenez-vous simplement de la notation. L'esperluette veut dire "adresse de". Donc `cout << &a;` se traduit en français par "Affiche l'adresse de la variable a".



On a déjà utilisé l'esperluette dans ce cours pour tout autre chose : lors de la déclaration d'une référence. C'est le même symbole qui est utilisé pour deux choses différentes. Attention à ne pas vous tromper !

Voyons maintenant ce que l'on peut faire avec ces adresses.

Les pointeurs

Les adresses sont des nombres. Vous connaissez plusieurs types permettant de stocker des nombres : `int`, `unsigned int`, `double`. Peut-on donc stocker une adresse dans une variable ?

La réponse est "oui". C'est possible. Mais pas avec les types que vous connaissez. Il nous faut utiliser un type un peu particulier : le **pointeur**.

Un pointeur est une variable qui contient l'adresse d'une autre variable.

Retenez bien cette phrase. Elle peut vous sauver la vie dans les moments les plus difficiles de ce chapitre. 😊

Déclarer un pointeur

Pour déclarer un pointeur, il faut, comme pour les variables, deux choses :

- Un type
- Un nom

Pour le nom, il n'y a rien de particulier à signaler. Les mêmes règles que pour les variables s'appliquent. Ouf !
Le type d'un pointeur a une petite subtilité. Il faut indiquer quel est le type de variable dont on veut stocker l'adresse et ajouter une étoile (*). 🤔 Je crois qu'un exemple sera plus simple.

Code : C++ - Déclaration d'un pointeur

```
int *pointeur;
```

Ce code déclare un pointeur qui peut contenir l'adresse d'une variable de type `int`.



On peut également écrire `int* pointeur` (l'étoile collée au mot `int`). Cette notation a un léger désavantage, c'est qu'on ne peut pas déclarer plusieurs pointeurs sur la même ligne comme ceci : `int* pointeur1, pointeur2, pointeur3;`. En faisant ça, seul `pointeur1` sera un pointeur, les deux autres variables seront des entiers tout à fait standards.

On peut bien sûr faire ça pour n'importe quel type :

Code : C++ - Déclaration de pointeurs

```
double *pointeurA; //Un pointeur qui peut contenir l'adresse d'un
nombre a virgule

unsigned int *pointeurB; //Un pointeur qui peut contenir l'adresse
d'un nombre entier positif

string *pointeurC; //Un pointeur qui peut contenir l'adresse d'une
chaîne de caractères

vector<int> *pointeurD; //Un pointeur qui peut contenir l'adresse
d'un tableau dynamique de nombres entiers

int const *pointeurE; //Un pointeur qui peut contenir l'adresse d'un
nombre entier constant
```

Pour le moment, ces pointeurs ne contiennent aucune adresse connue. C'est une situation très dangereuse. Si vous essayez d'utiliser le pointeur, vous ne savez pas quelle case de la mémoire vous manipulez. Ça peut être n'importe laquelle, par exemple la case qui contient votre mot de passe Windows ou la case qui contient l'heure actuelle. J'imagine que vous vous rendez compte des conséquences que peut avoir une mauvaise manipulation des pointeurs. Il ne faut donc **JAMAIS** déclarer un pointeur sans lui donner d'adresse.

Il faut donc toujours déclarer un pointeur en lui donnant la valeur 0 pour être tranquille :

Code : C++ - Déclaration d'un pointeur

```
int *pointeur(0);
double *pointeurA(0);
unsigned int *pointeurB(0);
string *pointeurC(0);
vector<int> *pointeurD(0);
int const *pointeurE(0);
```

Vous l'avez peut-être remarqué sur mon schéma un peu plus tôt, la première case de la mémoire avait l'adresse 1. En effet,

l'adresse 0 n'existe pas.

Lorsque vous créez un pointeur contenant l'adresse 0, cela signifie qu'il ne contient l'adresse d'**aucune** case.



Je me répète, mais c'est très important : déclarez toujours vos pointeurs en les initialisant à l'adresse 0.

Stocker une adresse

Maintenant qu'on a la variable, il n'y a plus qu'à mettre une valeur dedans. Vous savez déjà comment obtenir l'adresse d'une variable (rappelez-vous du **&**). Allons-y !

Code : C++

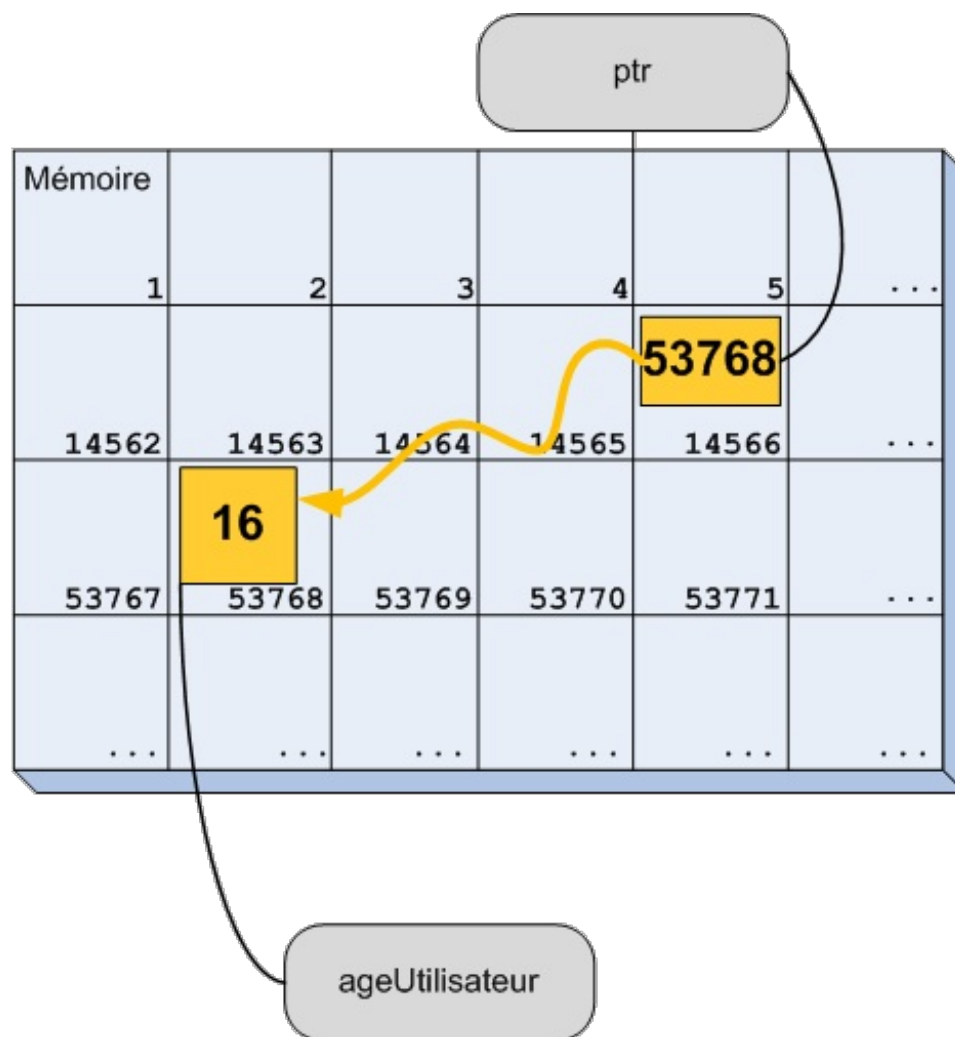
```
int main()
{
    int ageUtilisateur(16);    //Une variable de type int.
    int *ptr(0);               //Un pointeur pouvant contenir
    l'adresse d'un nombre entier.

    ptr = &ageUtilisateur;    //On met l'adresse de 'ageUtilisateur'
    dans le pointeur 'ptr'.

    return 0;
}
```

La ligne 6 est celle qui nous intéresse. Elle écrit l'adresse de la variable `ageUtilisateur` dans le pointeur `ptr`. On dit alors que le pointeur `ptr` **pointe sur** `ageUtilisateur`.

Voyons comment tout cela se déroule dans la mémoire avec un ... schéma ! 😊



On retrouve quelques éléments familiers. La mémoire avec sa grille de cases et la variable `ageUtilisateur` dans la case n°53768.

La nouveauté est bien sûr le pointeur. Dans la case mémoire n°14566, il y a une variable nommée `ptr` qui a pour valeur l'adresse 53768, c'est-à-dire l'adresse de la variable `ageUtilisateur`.

Voilà. Vous savez tout ou presque. Cela peut sembler absurde pour le moment ("Pourquoi stocker l'adresse d'une variable dans une autre case ?"), mais faites-moi confiance les choses vont progressivement s'éclaircir pour vous.

Si vous avez compris le schéma précédent, alors vous pouvez vous attaquer aux programmes les plus complexes.

Afficher l'adresse

Comme pour toutes les variables, on peut afficher le contenu d'un pointeur.

Code : C++

```
#include <iostream>
using namespace std;

int main()
{
    int ageUtilisateur(16);
    int *ptr(0);

    ptr = &ageUtilisateur;

    cout << "L'adresse de 'ageUtilisateur' est : " <<
    &ageUtilisateur << endl;
    cout << "La valeur de pointeur est : " << ptr << endl;
```

```
    return 0;  
}
```

Ce qui donne :

Code : Console

```
L'adresse de 'ageUtilisateur' est : 0x2ff18  
La valeur de pointeur est : 0x2ff18
```

La valeur du pointeur est donc bien l'adresse de la variable pointée. On a bien réussi à stocker une adresse ! 😊

Accéder à la valeur pointée

Vous vous souvenez du but des pointeurs ? Accéder à une variable sans passer par son nom. Voici comment faire. Il faut utiliser l'étoile (*) sur le pointeur pour afficher la valeur de la variable pointée.

Code : C++

```
int main()  
{  
    int ageUtilisateur(16);  
    int *ptr(0);  
  
    ptr = &ageUtilisateur;  
  
    cout << "La valeur est : " << *ptr << endl;  
  
    return 0;  
}
```

En faisant `cout << *ptr`, le programme va effectuer les étapes suivantes :

1. Aller dans la case mémoire nommée `ptr`.
2. Lire la valeur enregistrée.
3. "Suivre la flèche" pour aller à l'adresse pointée.
4. Lire la valeur stockée dans la case.
5. Afficher cette valeur. Ici, ce sera bien sûr 16 qui sera affiché.

En utilisant l'étoile, on accède à la **valeur de la variable pointée**. C'est ce qui s'appelle **déréférencer** un pointeur. Voici donc un deuxième moyen d'accéder à la valeur de `ageUtilisateur`.



Mais, à quoi cela sert-il ? 🤔

Je suis sûr que vous vous êtes retenu de poser la question avant. 😊 C'est vrai que ça a l'air assez inutile. Eh bien, je ne peux pas vous répondre rapidement pour le moment. 🤔

Il va falloir lire la fin de ce chapitre pour tout savoir.

Récapitulatif de la notation

Je suis d'accord avec vous, la notation est compliquée. L'étoile a deux significations différentes et on utilise l'esperluette alors qu'elle est déjà utilisée pour les références... Ce n'est pas ma faute ! Si vous voulez vous plaindre, il faut voir du côté des concepteurs du langage. C'est eux les responsables de ce charabia. 🤖

Nous, on ne peut que faire avec. Essayons donc de récapituler le tout.

Pour une variable `int` `nombre` :

- `nombre` permet d'accéder à la **valeur** de la variable.
- `&nombre` permet d'accéder à l'**adresse** de la variable.

Sur un pointeur `int` `*pointeur` :

- `pointeur` permet d'accéder à la **valeur** du pointeur, c'est-à-dire à l'**adresse de la variable pointée**.
- `*pointeur` permet d'accéder à la **valeur de la variable pointée**.

C'est ce qu'il faut retenir de cette sous-partie. Je vous invite à tester tout ça chez vous pour bien vérifier que vous avez compris comment afficher une adresse, comment utiliser un pointeur, etc.

"C'est en forgeant qu'on devient forgeron" dit le dicton, eh bien c'est en programmant avec des pointeurs que l'on devient programmeur. Il faut impérativement s'entraîner pour bien comprendre. Les meilleurs sont tous passés par là et je peux vous assurer qu'ils ont aussi souffert en découvrant les pointeurs. Si vous ressentez une petite douleur dans la tête, prenez un cachet d'aspirine, faites une pause, puis relisez ce que vous venez de lire encore et encore. Aidez-vous en particulier des schémas ! 😊

L'allocation dynamique

Vous vouliez savoir à quoi servent les pointeurs ? Vous êtes sûr ? Bon, alors je vous montre une première utilisation.

La gestion automatique de la mémoire

Dans notre tout premier chapitre sur les variables, je vous avais expliqué que lors de la déclaration d'une variable, le programme effectuait deux étapes :

1. Il demande à l'ordinateur de lui fournir une zone dans la mémoire. En termes techniques, on parle d'**allocation** de la mémoire.
2. Il remplit cette case avec la valeur fournie. On parle alors d'**initialisation** de la variable.

Tout cela est entièrement automatique, le programme se débrouille tout seul. De même lorsque l'on arrive à la fin d'une fonction, le programme rend la mémoire utilisée à l'ordinateur. C'est ce qu'on appelle la **libération** de la mémoire. C'est à nouveau automatique. Nous n'avons jamais dû dire à l'ordinateur : "Tiens reprends cette case mémoire, je n'en ai plus besoin."

Tout ceci se faisait automatiquement. Nous allons maintenant apprendre à le faire manuellement, et pour cela... vous vous doutez sûrement qu'on va utiliser les pointeurs. 😊

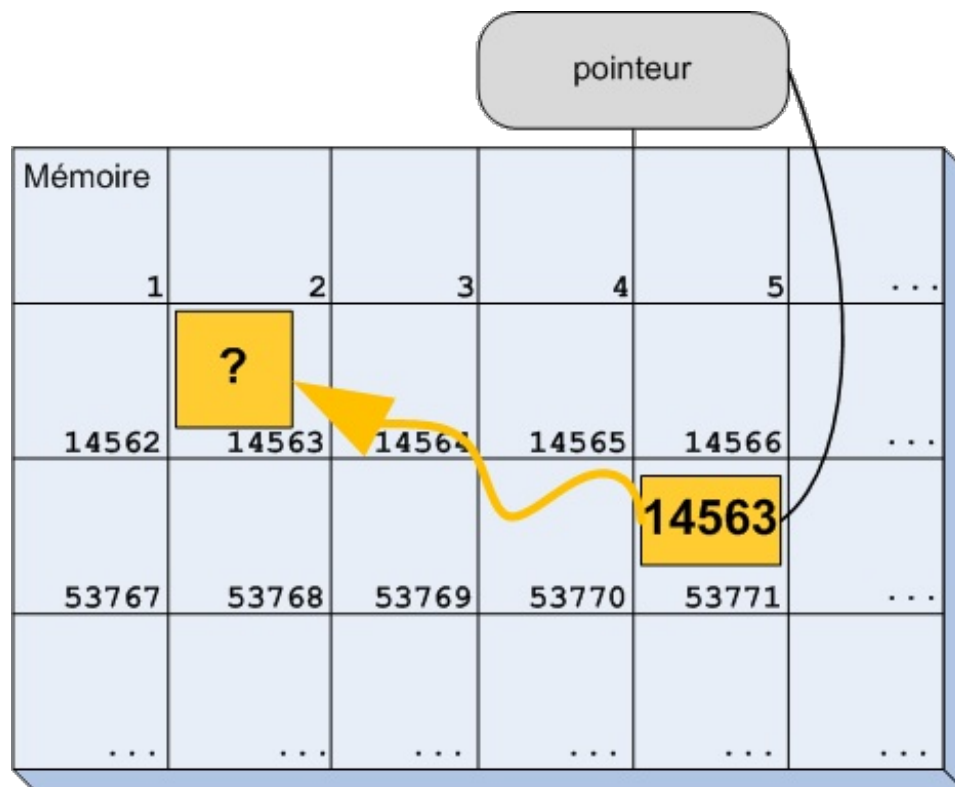
Allouer un espace mémoire

Pour demander manuellement une case dans la mémoire, il faut utiliser l'opérateur `new`. `new` va demander une case à l'ordinateur et renvoyer un **pointeur** pointant vers cette case.

Code : C++

```
int *pointeur(0);  
pointeur = new int;
```

La deuxième ligne demande une case mémoire pouvant stocker un entier et l'adresse de cette case est stockée dans le pointeur. Le mieux est encore de prendre un petit schéma.



Ce schéma est très similaire au précédent. Il y a deux cases mémoires utilisées :

- La case 14563 qui contient une variable de type `int` non-initialisée.
- La case 53771 qui contient un pointeur pointant sur la variable.

Rien de neuf. Mais, la chose importante, c'est que la variable à la case 14563 n'a **pas** d'étiquette. Le seul moyen d'y accéder est donc de passer par le pointeur.



Si vous changez la valeur du pointeur, vous perdez le seul moyen d'accès à la case mémoire. Vous ne pourrez donc plus l'utiliser ni la supprimer ! Elle sera définitivement perdue mais elle continuera à prendre de la place. C'est ce qu'on appelle une **fuite de mémoire**.

Il faut donc faire très attention ! 😬

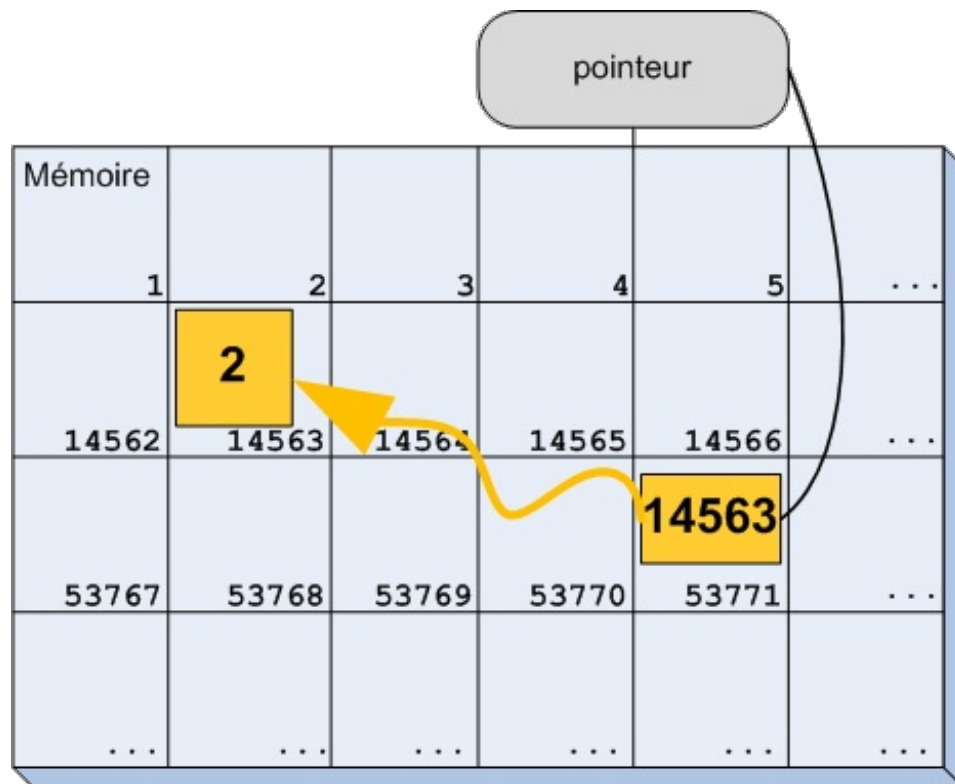
Une fois allouée manuellement, la variable s'utilise comme n'importe quelle autre. On doit juste se rappeler qu'il faut y accéder par le pointeur en le déréférençant.

Code : C++

```
int *pointeur(0);
pointeur = new int;

*pointeur = 2; //On accède à la case mémoire pour en modifier la
valeur
```

La case sans étiquette est maintenant remplie. La mémoire est donc dans l'état suivant :



A part son accès un peu spécial (via `*pointeur`), nous avons donc une variable en tout point semblable à une autre.

Il nous faut maintenant rendre la mémoire que l'ordinateur nous a gentiment prêtée.

Libérer la mémoire

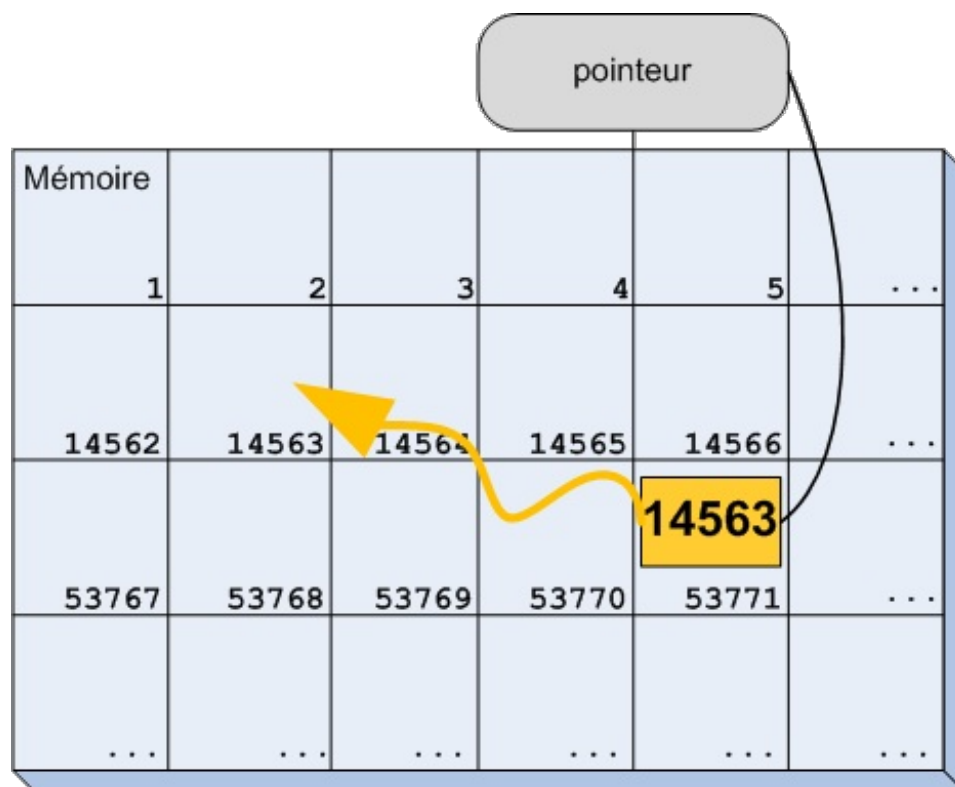
Une fois que l'on a plus besoin de la case mémoire, il faut la rendre à l'ordinateur. Cela se fait via l'opérateur `delete`.

Code : C++

```
int *pointeur(0);
pointeur = new int;

delete pointeur;    //On libère la case mémoire
```

La case est alors rendue à l'ordinateur qui pourra l'utiliser pour autre chose. Le pointeur, lui, existe toujours. Et il pointe toujours sur la case mais vous n'avez **plus le droit** de l'utiliser.



L'image est très parlante. Si l'on suit la flèche, on arrive sur une case qui ne nous appartient pas. Il faut donc impérativement empêcher ça. Imaginez que cette case soit soudainement utilisée par un autre programme ! Vous risqueriez de modifier les variables de cet autre programme.

Il est donc **essentiel** de supprimer cette "flèche" en mettant le pointeur à l'adresse 0 après avoir utilisé **delete**. Ne pas le faire est une source très courante de plantage des programmes.

Code : C++

```
int *pointeur(0);
pointeur = new int;

delete pointeur;    //On libère la case mémoire
pointeur = 0;       //On indique que le pointeur ne pointe vers
                    plus rien
```



N'oubliez pas de libérer la mémoire. Si vous ne le faites pas, votre programme risque d'utiliser de plus en plus de mémoire jusqu'au moment où il n'y aura plus aucune case disponible ! Votre programme va alors planter.

Un exemple complet

Terminons cette section avec un exemple complet : un programme qui demande son âge à l'utilisateur et qui l'affiche en utilisant un pointeur.

Code : C++

```
#include <iostream>
using namespace std;

int main()
{
    int* pointeur(0);
    pointeur = new int;
```

```
    cout << "Quel est votre age ? ";
    cin >> *pointeur; //On écrit dans la case mémoire pointée par
    le pointeur 'pointeur'

    cout << "Vous avez " << *pointeur << " ans." << endl; //On
    utilise à nouveau *pointeur

    delete pointeur; //Ne pas oublier de libérer la mémoire
    pointeur = 0; //Et de faire pointer le pointeur vers rien

    return 0;
}
```

Ce programme est plus compliqué que sa version sans allocation dynamique ! C'est vrai. Mais on a le contrôle complet sur l'allocation et la libération de la mémoire.

Dans la plupart des cas, ce n'est pas utile de le faire. Mais vous verrez plus tard que, pour faire des fenêtres, la bibliothèque Qt utilise beaucoup **new** et **delete**. On peut ainsi maîtriser précisément quand une fenêtre est ouverte et quand on la referme par exemple.

Quand utiliser des pointeurs

Je vous avais promis des explications sur quand utiliser des pointeurs. Les voici. 🤔

Il y a en réalité trois cas d'application :

- Gérer soi-même le moment de la création et de la destruction des cases mémoire.
- Partager une variable dans plusieurs morceaux du code.
- Sélectionner une valeur parmi plusieurs options.

Si vous n'êtes pas dans un de ces trois cas, c'est très certainement que vous n'avez pas besoin des pointeurs.

Vous connaissez déjà le premier de ces trois cas. Concentrons nous sur les deux autres.

Partager une variable

Pour l'instant, je ne peux pas vous donner un code source complet pour ce cas d'utilisation. Ou alors, il ne sera pas intéressant du tout. Quand vous aurez quelques notions de programmation orientée objet, vous aurez de vrais exemples.

En attendant, je vous propose un exemple plus ... visuel. 😊

Vous avez déjà joué à un jeu de stratégie ? J'imagine que oui, prenons un exemple tiré d'un jeu de ce genre. Voici une image issue du fameux *Warcraft III*.



Programmer un tel jeu est bien sûr très compliqué. Mais on peut quand même réfléchir à certains des mécanismes utilisés. Sur l'image, on voit des humains en rouge attaquer des orcs en bleu. Chaque personnage a une cible précise. Par exemple, le fusilier au milieu de l'écran semble tirer sur le gros personnage bleu qui tient une hache.

Nous verrons dans la suite de ce cours comment créer des objets, c'est-à-dire des variables plus évoluées. Par exemple une variable de type "personnage", de type "orc" ou encore de type "batiment". Bref, chaque élément du jeu pourra être modélisé en C++ par un objet.

Comment feriez-vous pour indiquer, en C++, la cible du personnage rouge ? 🤔

Bien sûr, vous ne savez pas encore comment faire en détail, mais vous avez peut-être une petite idée. Rappelez-vous du titre de ce chapitre. 😊

Oui oui, un pointeur est une bonne solution ! Chaque personnage possède un pointeur qui pointe vers sa cible. Il a ainsi un moyen de savoir qui viser et attaquer. On pourrait par exemple écrire quelque chose comme :

Code : C++

```
Personnage *cible; //Un pointeur qui pointe sur un autre personnage
```

Quand il n'y a pas de combat en cours, le pointeur pointe vers l'adresse 0, il n'a pas de cible. Quand le combat est engagé, le pointeur pointe vers un ennemi. Et finalement, quand cet ennemi meurt, on déplace le pointeur vers une autre adresse, c'est-à-dire vers un autre personnage.

Le pointeur est donc réellement utilisé ici comme une flèche reliant un personnage à son ennemi.

Nous verrons comment écrire du code comme cela dans la suite, je crois même que créer un mini-RPG sera le thème principal des chapitres de la partie II. Mais chut, c'est pour plus tard. 😊

Choisir parmi plusieurs éléments

Le troisième et dernier cas permet de faire évoluer un programme en fonction des choix de l'utilisateur.

Prenons le cas d'un QCM. Nous allons demander à l'utilisateur de choisir parmi trois réponses possibles à une question. Une fois qu'il aura choisi, nous allons utiliser un pointeur pour indiquer quelle réponse a été choisie.

Code : C++

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string reponseA, reponseB, reponseC;
    reponseA = "La peur des jeux de loterie";
    reponseB = "La peur du noir";
    reponseC = "La peur des vendredis treize";

    cout << "Qu'est-ce que la 'kenophobie' ? " << endl; //On pose la
question
    cout << "A) " << reponseA << endl; //Et on affiche les trois
propositions
    cout << "B) " << reponseB << endl;
    cout << "C) " << reponseC << endl;

    char reponse;
    cout << "Votre reponse (A,B ou C) : ";
    cin >> reponse; //On récupère la
réponse de l'utilisateur

    string *reponseUtilisateur(0); //Un pointeur qui
pointera sur la réponse choisie
    switch(reponse)
    {
        case 'A':
            reponseUtilisateur = &reponseA; //On déplace le pointeur
sur la réponse choisie
            break;
        case 'B':
            reponseUtilisateur = &reponseB;
            break;
        case 'C':
            reponseUtilisateur = &reponseC;
            break;
        default:
            cout << "Choix invalide" << endl;
            reponseUtilisateur = &reponseA; //Par défaut, on choisit A
            break;
    }

    //On peut alors utiliser le pointeur pour afficher la réponse
choisie
    cout << "Vous avez choisi la reponse : " << *reponseUtilisateur
<< endl;

    return 0;
}
```

Une fois que le pointeur a été déplacé (dans le **switch**) on peut l'utiliser comme moyen d'accès à la réponse de l'utilisateur. On a ainsi un moyen d'atteindre directement cette variable sans devoir refaire le test à chaque fois qu'on en a besoin. C'est une variable qui contient une valeur que l'on ne pouvait pas connaître avant (puisqu'elle dépend de ce que l'utilisateur a entré).

C'est certainement le cas d'utilisation le plus rare des trois, mais il arrive parfois qu'on soit dans cette situation. Il sera alors temps de vous rappeler des pointeurs ! 😊

Nous en avons fini avec les bases du C++ !

Vous n'avez peut-être pas tout compris dans les moindres détails, ce n'est pas grave ! C'est surtout en pratiquant que l'on apprend et je vous assure que de nombreuses personnes (dont moi 😊) ont eu besoin de beaucoup de temps pour bien tout saisir. Certaines des notions présentées jusque-là vont réapparaître plus loin dans le cours, ce sera alors le moment de venir lire ce qu'il vous manquait.

Vous êtes prêt ? Alors attaquons le monde magique de la *programmation orientée objet*, le coeur du C++... 🧙

Partie 2 : [Théorie] La Programmation Orientée Objet

Maintenant que vous connaissez les bases de la programmation C++, attaquons le coeur du sujet : la programmation orientée objet (POO) !

Soyez attentifs car les choses deviennent un peu plus complexes à partir d'ici. Prenez bien le temps de tout lire, car vous ne pouvez pas faire de C++ sans bien connaître la POO. 😊

Introduction : la vérité sur les strings enfin dévoilée

Nous attaquons la 2ème partie du cours de C++. Et comme dans la vie rien n'est jamais simple, cette "deuxième moitié" sera la plus dense et... la plus délicate aussi. 😊

Nous allons maintenant, et dans les chapitres suivants, découvrir la notion de **programmation orientée objet** (POO). Comme je vous l'ai dit plus tôt, c'est une nouvelle façon de programmer. Ça ne va pas révolutionner immédiatement vos programmes, ça va vous paraître un peu inutile au début, mais faites-moi confiance : faites l'effort de faire ce que je dis à la lettre, et bientôt vous trouverez cette manière de faire bien plus naturelle. Vous saurez plus aisément comment *organiser* vos programmes.

Ce chapitre va vous parler des 2 facettes de la POO, le côté *utilisateur* et le côté *créateur*.

Puis, je vais faire carrément l'inverse de ce que tous les cours de programmation font (je sais je suis fou 😜) : au lieu de commencer par vous apprendre à *créer* des objets, je vais d'abord vous montrer comment les *utiliser* avec pour exemple le type **string** fourni par le langage C++.

Des objets... pour quoi faire ?

Ils sont beaux, ils sont frais mes objets

S'il y a bien un mot qui doit vous frustrer depuis que vous en entendez parler, c'est celui-ci : **objet**.



Encore un concept mystique ? Un délire de programmeurs après une soirée trop arrosée ?

Non parce que franchement, un objet c'est quoi ? Mon écran est un objet, ma voiture est un objet, mon téléphone portable... ce sont tous des objets !

Bien vu, c'est un premier point. 😊

En effet, nous sommes entourés d'objets. En fait, tout ce que nous connaissons (ou presque) peut être considéré comme un objet. L'idée de la programmation orientée objet, c'est de manipuler des éléments que l'on appelle des "objets" dans son code source.

Voici quelques exemples d'objets dans des programmes courants :

- Une fenêtre
- Un bouton
- Un personnage de jeu vidéo
- Une musique

Comme vous le voyez, beaucoup de choses peuvent être considérées comme des objets. 😊



Mais concrètement, c'est quoi ? Une variable ? Une fonction ?

Ni l'un, ni l'autre. C'est un nouvel élément en programmation.

Pour être plus précis, un objet c'est... un mélange de plusieurs variables et fonctions. 😊

Ne faites pas cette tête-là, vous allez découvrir tout cela par la suite. 😊

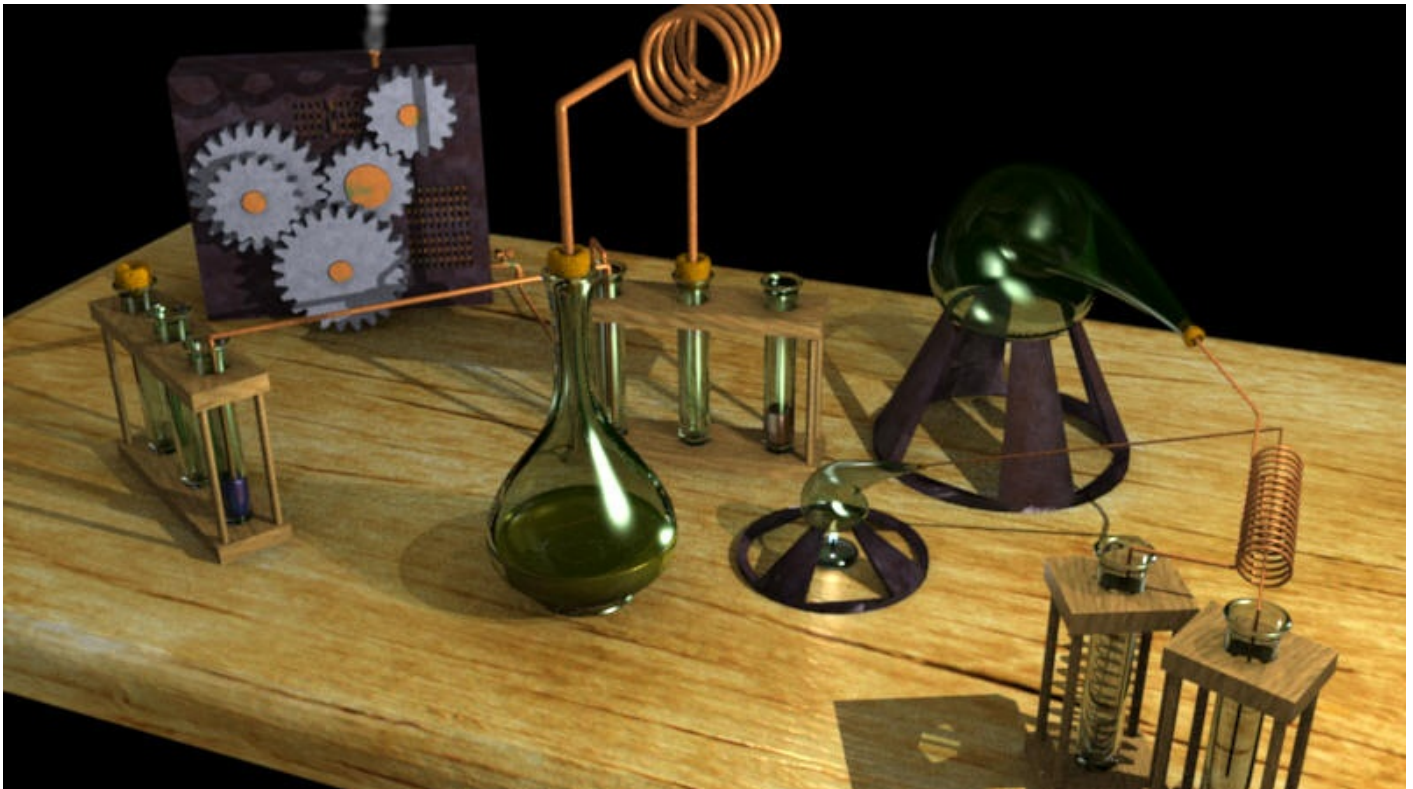
Imaginez... un objet

Pour éviter que ce que je vous raconte ressemble à un traité d'art moderne conceptuel, on va imaginer ensemble ce qu'est un objet à l'aide de plusieurs schémas concrets.

Les schémas 3D que vous allez voir par la suite ont été réalisés pour moi par l'ami Nab, que je remercie d'ailleurs vivement au passage.

Imaginez qu'un programmeur décide un jour de créer un programme qui permet d'afficher une fenêtre à l'écran, de la redimensionner, de la déplacer, de la supprimer... Le code est complexe : il va avoir besoin de plusieurs fonctions qui s'appellent entre elles, et de variables pour mémoriser la position, la taille de la fenêtre, etc.

Il met du temps à écrire ce code, c'est un peu compliqué, mais il y arrive. Au final, le code qu'il a écrit est composé de plusieurs fonctions et variables. Quand on regarde ça pour la première fois, ça ressemble à une expérience de savant fou à laquelle on ne comprend rien :



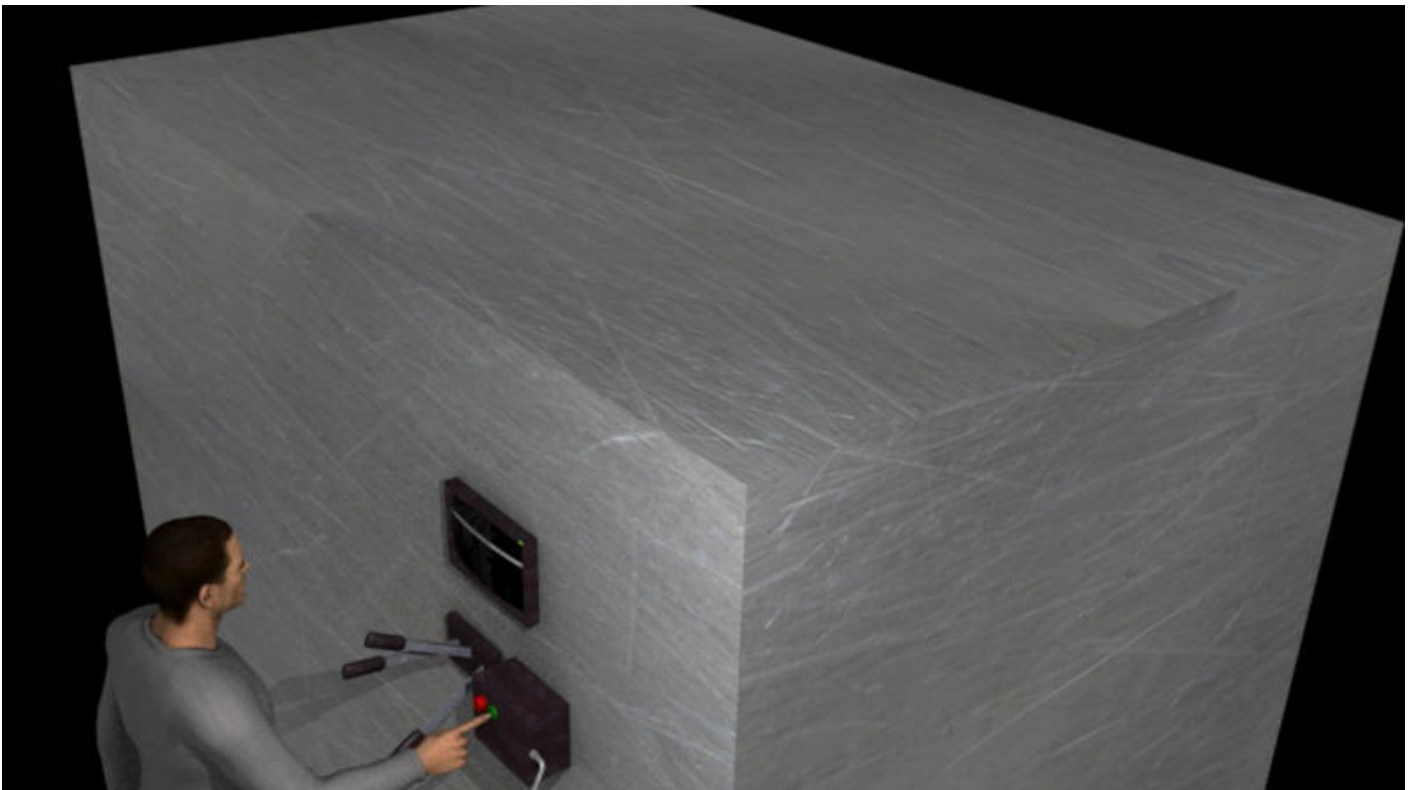
Ce programmeur est content de son code et veut le distribuer sur internet pour que tout le monde puisse créer des fenêtres sans passer du temps à tout réécrire. Seulement voilà, à moins d'être un expert en chimie certifié, vous allez mettre pas mal de temps avant de comprendre comment tout ce bazar fonctionne.

Quelle fonction appeler en premier ? Quelles valeurs envoyer à quelle fonction pour redimensionner la fenêtre ? Autrement dit : comment utiliser ce bazar sans qu'une fiole ne nous explose entre les mains ? 🤖

C'est là que notre ami programmeur pense à nous. Il conçoit son code de manière orientée objet. Cela signifie qu'il place tout son bazar chimique à l'intérieur d'un simple cube. Ce cube est ce qu'on appelle un objet :



Ici, une partie du cube a été volontairement mise en transparence pour vous montrer que nos fioles chimiques sont bien situées à l'intérieur du cube. Mais en réalité, le cube est complètement opaque, on ne voit **rien** de ce qu'il y a à l'intérieur :



Ce cube contient toutes les fonctions et les variables (nos fioles de chimie), mais il les masque à l'utilisateur.

Au lieu d'avoir des tonnes de tubes et fioles chimiques dont il faut comprendre le fonctionnement, on nous propose juste quelques boutons sur la face avant du cube : un bouton "ouvrir fenêtre", un bouton "redimensionner", etc. L'utilisateur n'a plus qu'à se servir des boutons du cube et n'a plus besoin de se soucier de tout ce qui se passe à l'intérieur. Pour l'utilisateur, c'est donc complètement simplifié.

En clair : programmer de manière orientée objet, c'est *créer* du code source (peut-être complexe), mais que l'on *masque* en le plaçant à l'intérieur d'un cube (un objet) à travers lequel on ne voit rien. Pour le programmeur qui va *l'utiliser*, travailler avec un objet est donc beaucoup plus simple qu'avant : il a juste à appuyer sur des boutons et n'a pas besoin d'être diplômé en chimie pour s'en servir.

Bien sûr, c'est une image, mais c'est ce qu'il faut comprendre et retenir pour le moment. 😊

Nous n'allons pas voir tout de suite comment faire pour *créer* des objets. En revanche, nous allons apprendre à en *utiliser* un. Nous allons nous pencher sur le cas de `string` dans ce chapitre.



J'ai déjà utilisé le type `string`, ce n'est pas une nouveauté pour moi ! C'est le type qui permet de stocker du texte en mémoire c'est ça ?

Oui. Mais comme je vous l'ai dit il y a quelques chapitres, le type `string` est différent des autres. `int`, `bool`, `float`, `double` sont des types naturels du C++. Ils stockent des données très simples. Ce n'est pas le cas de `string` qui est en fait... un objet ! Le type `string` cache beaucoup de secrets à l'intérieur de sa boîte.

Jusqu'ici, nous n'avons fait qu'appuyer sur des boutons (comme sur les schémas), mais en réalité ce qui se cache à l'intérieur de la boîte des objets `string` est très complexe. Horriblement complexe. 😬

L'horrible secret du type `string`

Les enfants, il faut que je vous raconte une terrible vérité : ~~le Père Noël n'existe pas~~ ! Euh non... mauvaise fiche désolé. Ah oui, voilà ce que je voulais dire : le type `string` est affreusement plus complexe qu'il n'en a l'air au fond de ses entrailles.

Grâce aux mécanismes de la programmation orientée objet, nous avons pu utiliser le type `string` dès les premiers chapitres de ce cours alors qu'il est pourtant assez compliqué dans son fonctionnement interne ! Pour vous en convaincre, je vais vous montrer comment fonctionne `string` "à l'intérieur du cube". Préparez-vous à d'horribles vérités. 😬



Pour un ordinateur, les lettres n'existent pas

Comme nous l'avons vu, l'avantage des objets est de masquer la complexité du code au programmeur. Plutôt que de manipuler des fioles chimiques dangereuses, ils nous permettent d'appuyer sur de simples boutons pour faire des choses parfois compliquées.

Et justement, les choses sont compliquées parce qu'à la base *un ordinateur ne sait pas gérer du texte* ! Oui, l'ordinateur n'est véritablement qu'une grosse machine à calculer dénuée de sentiments. Il ne reconnaît que des nombres.



Mais alors, si l'ordinateur ne peut manipuler que des nombres, comment se fait-il qu'il puisse afficher du texte à l'écran ?

C'est une vieille astuce que l'on utilise depuis longtemps. Peut-être avez-vous entendu parler de la **table ASCII** ? (prononcez "aski")

C'est une table qui sert de convention pour convertir des nombres en lettres.

Un extrait de la table ASCII

Nombre	Lettre	Nombre	Lettre
64	@	96	'
65	A	97	a

66	B	98	b
67	C	99	c
68	D	100	d
69	E	101	e
70	F	102	f
71	G	103	g
72	H	104	h
73	I	105	i
74	J	106	j
75	K	107	k
76	L	108	l
77	M	109	m

Comme vous le voyez, la lettre "A" majuscule correspond au nombre 65. La lettre "a" minuscule correspond au nombre 97, etc. Tous les caractères utilisés en anglais sont dans cette table. C'est pour ça que les caractères accentués ne sont pas utilisables de base en C++, ils n'apparaissent pas dans la table ASCII.



Cela veut dire qu'à chaque fois que l'ordinateur voit le nombre 65, il prend ça comme la lettre A ?

Non, l'ordinateur ne traduit un nombre en lettre que si on le lui demande. En pratique, on va se baser sur le type de la variable pour savoir si le nombre stocké est véritablement un nombre ou en fait une lettre :

- Si on utilise le type `int` pour stocker le nombre 65, l'ordinateur considérera que c'est un nombre.
- En revanche, si on utilise le type `char` pour stocker le nombre 65, l'ordinateur se dira "C'est la lettre A". Le type `char` (abréviation de *character*, "caractère" en français) est prévu pour stocker un caractère.

Le type `char` stocke donc un nombre qui est interprété comme un caractère.



Un `char` ne peut stocker qu'un seul caractère ? Comment fait-on alors pour stocker une phrase entière ?

Eh bien là non plus ce n'est pas simple ! C'est un autre problème que l'on va voir...

Les textes sont des tableaux de `char`

Puisque `char` ne peut stocker qu'une seule lettre, les programmeurs ont eu l'idée de créer... un tableau de `char` ! Les tableaux permettant de retrouver plusieurs variables d'un même type côte à côte en mémoire, ils sont le moyen idéal de stocker du texte (on parle aussi de "chaînes de caractères", vous comprenez maintenant pourquoi).

Ainsi, il suffit de déclarer un tableau de `char` comme ceci :

Code : C++

```
char texte[100];
```

... pour pouvoir stocker du texte (environ 100 caractères) !

Le texte n'est donc en fait qu'un assemblage de lettres en mémoire dans un tableau :

T	e	x	t	e
---	---	---	---	---

Chaque case correspond à un `char`. Tous ces `char` mis côte à côte forment du texte.



Attention : il faut prévoir suffisamment de place dans le tableau pour stocker tout le texte ! Ici, c'est un tableau de 100 cases, mais ça peut être juste si on veut stocker plusieurs phrases en mémoire !
Pour résoudre ce problème, on peut créer un très grand tableau (en prévision de la taille de ce qu'on va stocker), mais cela risque parfois de consommer beaucoup de mémoire pour rien.

Créer et utiliser des objets string

Vous venez d'en avoir un aperçu : gérer du texte n'est pas vraiment simple. Il faut créer un tableau de `char` dont chaque case correspond à un caractère, il faut prévoir une taille suffisante pour stocker le texte que l'on souhaite sinon ça plante... Bref, ça fait beaucoup de choses auxquelles il faut penser.

Ca ne vous rappelle pas nos fioles chimiques ? Eh oui, tout ceci est aussi dangereux et compliqué qu'une expérience de chimiste. C'est là que la programmation orientée objet intervient : un développeur place le tout dans un cube facile à utiliser où il suffit d'appuyer sur des boutons. **Ce cube, c'est l'objet `string`.**

Créer un objet string

La création d'un objet ressemble beaucoup à la création d'une variable classique comme `int` ou `double` :

Code : C++

```
#include <iostream>
#include <string> // Obligatoire pour pouvoir utiliser les objets
string

using namespace std;

int main()
{
    string maChaine; // Création d'un objet "maChaine" de type
string
    return 0;
}
```

Vous remarquerez pour commencer qu'il est nécessaire d'inclure le header de la librairie `string` pour pouvoir utiliser des objets de type `string` dans le code. 😊 C'est ce que j'ai fait à la 2ème ligne.

Intéressons-nous maintenant à la ligne où je crée un objet de type `string`...



Donc... on crée un objet de la même manière qu'on crée une variable ?

Il y a plusieurs façons de créer un objet, celle que vous venez de voir est la plus simple. Et, oui, c'est exactement comme si on avait créé une variable !



Mais mais... comment on fait pour différencier les objets des variables ?

C'est bien tout le problème : variables et objets se ressemblent dans le code. Pour éviter la confusion, il y a des conventions (qu'on n'est pas obligé de suivre). La plus célèbre d'entre elles est la suivante :

- Le type des **variables** commence par une **minuscule** (ex : int)
- Le type des **objets** commence par une **majuscule** (ex : Véture)

Je sais ce que vous allez me dire : "*string ne commence pas par une majuscule alors que c'est un objet !*". Il faut croire que ceux qui ont créé string ne respectaient pas cette convention. Mais rassurez-vous, maintenant la plupart des gens mettent une majuscule au début de leurs objets (dont moi), ça ne sera pas la foire dans la suite de ce cours. 😊

Initialiser la chaîne lors de la déclaration

Pour initialiser notre objet au moment de la déclaration (et donc lui donner une valeur !), il y a plusieurs possibilités. La plus courante consiste à ouvrir des parenthèses comme nous l'avons fait jusqu'ici :

Code : C++

```
int main()
{
    string maChaine("Bonjour !"); // Création d'un objet "maChaine"
    de type string et initialisation

    return 0;
}
```

C'est la technique classique que l'on connaît déjà, et qui s'applique aussi bien aux variables qu'aux objets. On dit que l'on *construit* l'objet.



Et comme pour les variables, il faut noter qu'il est aussi possible d'initialiser avec le signe égal : `string maChaine = "Bonjour !";`

On a maintenant créé un objet `maChaine` qui contient la chaîne "Bonjour !".
On peut l'afficher comme n'importe quelle chaîne de caractères avec un `cout` :

Code : C++

```
int main()
{
    string maChaine("Bonjour !");
    cout << maChaine << endl; // Affichage du string comme si
    c'était une chaîne de caractères

    return 0;
}
```

Code : Console

```
Bonjour !
```

Affecter une valeur à la chaîne après déclaration

Maintenant que notre objet est créé, ne nous arrêtons pas là. Changeons le contenu de la chaîne après sa déclaration :

Code : C++

```
int main()
{
    string maChaine("Bonjour !");
    cout << maChaine << endl;

    maChaine = "Bien le bonjour !";
    cout << maChaine << endl;

    return 0;
}
```

Code : Console

```
Bonjour !
Bien le bonjour !
```



Pour changer le contenu d'une chaîne *après* sa déclaration, on doit obligatoirement utiliser le symbole "=".

Ca n'a l'air de rien, mais c'est là que la magie de la POO opère. Vous, l'utilisateur, vous avez appuyé sur un bouton pour dire "Je veux maintenant que la chaîne à l'intérieur change pour *Bien le bonjour !*". A l'intérieur de l'objet, des mécanismes (des fonctions) se sont activées lorsque vous avez fait ça. Ces fonctions ont vérifié entre autres s'il y avait de la place pour stocker la chaîne dans le tableau de `char`. Elles ont vu que non. Elles ont alors créé un nouveau tableau de `char`, suffisamment long cette fois, pour stocker la nouvelle chaîne. Et elles ont détruit l'ancien tableau qui ne servait plus à rien, tant qu'à faire.

Et permettez-moi de vous parler franchement : ce qui s'est passé à l'intérieur de l'objet, on s'en fout royalement 🤪

C'est bien là tout l'intérêt de la POO : l'utilisateur n'a pas besoin de comprendre comment ça marche à l'intérieur. On s'en moque que le texte soit stocké dans un tableau de `char`. L'objet est en quelque sorte intelligent et gère tous les cas. Nous, on ne fait que l'utiliser ici.

Concaténation de chaînes

Imaginez que l'on souhaite concaténer (assembler) 2 chaînes. En théorie c'est compliqué à faire car il faut fusionner 2 tableaux de `char`. En pratique, la POO nous permet de ne pas avoir à nous soucier du fonctionnement interne :

Code : C++

```
int main()
{
    string chaine1("Bonjour !");
    string chaine2("Comment allez-vous ?");
    string chaine3;

    chaine3 = chaine1 + chaine2; // 3... 2... 1...
    Concaténatioooooooooo
    cout << chaine3 << endl;

    return 0;
}
```

Code : Console

```
Bonjour !Comment allez-vous ?
```

Ah, allez je reconnais, il manque un espace au milieu. On n'a qu'à changer la ligne de la concaténation :

Code : C++

```
chaine3 = chaine1 + " " + chaine2;
```

Résultat :

Code : Console

```
Bonjour ! Comment allez-vous ?
```

C'est très simple à utiliser, alors que derrière les fioles chimiques s'activent pour assembler les 2 tableaux de `char`.

Comparaison de chaînes

Vous en voulez encore ? Très bien !

Sachez que l'on peut comparer des chaînes entre elles à l'aide des symboles `==` ou `!=` (que l'on peut donc utiliser dans un `if` !).

Code : C++

```
int main()
{
    string chaine1("Bonjour !");
    string chaine2("Comment allez-vous ?");

    if (chaine1 == chaine2) // Faux
    {
        cout << "Les chaines sont identiques" << endl;
    }
    else
    {
        cout << "Les chaines sont differentes" << endl;
    }

    return 0;
}
```

Code : Console

```
Les chaines sont differentes
```

A l'intérieur de l'objet, la comparaison se fait caractère par caractère entre les deux tableaux de `char` (à l'aide d'une boucle qui compare chacune des lettres). Nous, nous n'avons pas à nous soucier de tout cela : nous demandons à l'objet `chaine1` s'il est

identique à `chaine2`, il fait des calculs et nous répond très simplement par un oui ou un non. 😊

Opérations sur les string

Le type `string` ne s'arrête pas à ce que nous venons de voir. Comme tout bon objet qui se respecte, il propose un nombre important d'autres fonctionnalités qui permettent de faire tout ce dont on a besoin.

Nous n'allons pas passer toutes les fonctionnalités des `string` en revue (elles sont pas toutes indispensables et ce serait un peu long). Nous allons voir les principales dont vous pourriez avoir besoin dans la suite du cours 😊

Attributs et méthodes

Je vous avais dit qu'un objet était constitué de variables et de fonctions. En fait, on en reparlera plus tard mais le vocabulaire est un peu différent avec les objets. Les variables contenues à l'intérieur des objets sont appelées **attributs**, et les fonctions sont appelées **méthodes**.

Imaginez que chaque méthode (fonction) que propose un objet correspond à un bouton différent sur la façade avant du cube. 😊



On parle aussi de "variables membres" et de "fonctions membres".

Pour appeler la méthode d'un objet, on utilise une écriture que vous avez déjà vue :

objet.methode()

On sépare le nom de l'objet et le nom de la méthode par un point. Cela signifie "Sur l'objet indiqué, j'appelle cette méthode" (traduction : "sur le cube indiqué, j'appuie sur ce bouton pour déclencher une action").



En théorie, on peut aussi accéder aux variables membres (les "attributs") de l'objet de la même manière. Cependant, en POO, il y a une règle très importante qui dit que l'utilisateur ne doit pas pouvoir accéder aux variables membres, mais seulement aux fonctions membres (les méthodes). On en reparlera dans le prochain chapitre plus en détail.

Quelques méthodes utiles du type string

La méthode `size()`

La méthode `size()` permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type `string`.

Cette méthode ne prend aucun paramètre et renvoie la longueur de la chaîne. Comme vous venez de le découvrir, il va falloir appeler la méthode de la manière suivante :

Code : C++

```
maChaine.size()
```

Essayons ça dans un code complet qui affiche la longueur de la chaîne :

Code : C++

```
int main()
{
    string maChaine("Bonjour !");
```



```
cout << "Longueur de la chaine : " << maChaine.size();  
  
return 0;  
}
```

Code : Console

Longueur de la chaine : 9

La méthode erase()

Cette méthode très simple supprime tout le contenu de la chaîne :

Code : C++

```
int main()  
{  
    string chaine("Bonjour !");  
    chaine.erase();  
    cout << "La chaine contient : " << chaine << endl;  
  
    return 0;  
}
```

Code : Console

La chaine contient :

Comme on pouvait s'y attendre, la chaîne ne contient plus rien 😊

Notez que c'est équivalent à faire :

**Code : C++**

```
chaine = "";
```

La méthode substr()

Une autre méthode qui peut s'avérer utile : substr(). Elle permet de ne prendre qu'une partie de la chaîne stockée dans un string.



substr signifie "substring", soit "sous-chaîne" en anglais.

Tenez, on va regarder son prototype, vous allez voir que c'est intéressant :

Code : C++

```
string substr( size_type index, size_type num = npos );
```

Cette méthode retourne donc un objet de type string. Ce sera la sous-chaîne après "découpage".

Elle prend 2 paramètres, ou plus exactement : 1 paramètre obligatoire, 1 paramètre facultatif. En effet, *num* possède une valeur par défaut (npos) ce qui fait que le second paramètre ne doit pas obligatoirement être renseigné.

- **index** permet d'indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère)
- **num** permet d'indiquer le nombre de caractères que l'on prend. Par défaut, la valeur est npos, ce qui correspond à prendre tous les caractères qui restent. Si vous indiquez 2, la méthode ne renverra que 2 caractères.

Allez, un exemple sera plus parlant je crois 😊

Code : C++

```
int main()
{
    string chaine("Bonjour !");
    cout << chaine.substr(3) << endl;

    return 0;
}
```

Code : Console

```
jour !
```

On a demandé à couper à partir du 3ème caractère (soit la lettre "j" vu que la première lettre correspond au caractère n°0). On a volontairement omis le second paramètre facultatif, ce qui fait que du coup `substr()` a renvoyé tous les caractères restants avant la fin de la chaîne. Essayons de renseigner le paramètre facultatif pour ne pas prendre le point d'exclamation par exemple :

Code : C++

```
int main()
{
    string chaine("Bonjour !");
    cout << chaine.substr(3, 4) << endl;

    return 0;
}
```

Code : Console

```
jour
```

Bingo ! 😊

On a demandé à prendre 4 caractères en partant du caractère n°3, ce qui fait qu'on a récupéré "jour" 😊

Comme nous l'avions vu dans le chapitre sur les tableaux, il existe une autre manière de faire pour accéder à un seul

caractère. On utilise les crochets [] comme pour les tableaux :

Code : C++



```
string chaine("Bonjour !");  
cout << chaine[3] << endl; //Affiche la lettre 'j'
```

On utilise `substr()` que si l'on a besoin d'accéder à plus d'une lettre à la fois.

La méthode `c_str()`

Celle-là est un peu particulière, mais parfois fort utile. Son rôle ? Retourner un pointeur vers le tableau de `char` que contient l'objet de type `string`.

Quel intérêt me direz-vous ? En C++, à priori aucun intérêt. On préfère largement manipuler un objet `string` plutôt qu'un tableau de `char` car c'est plus simple et plus sûr.

Néanmoins, il peut (j'ai bien dit il "peut") arriver que vous deviez envoyer à une fonction un tableau de `char`. Dans ce cas, la méthode `c_str()` vous permet de récupérer l'adresse du tableau de `char` qui se trouve à l'intérieur de l'objet `string`. Nous en avons eu besoin pour indiquer le nom du fichier à ouvrir dans un chapitre précédent, souvenez-vous :

Code : C++

```
string const nomFichier("C:/Nanoc/scores.txt");  
ofstream monFlux(nomFichier.c_str());
```

L'usage de `c_str()` reste assez rare malgré tout.

Comme le disait si bien ma prof d'informatique "C'est plus confortable de travailler avec un `string`" (je vous jure que c'est vrai, j'étais là 😊)

Bon plus sérieusement 😊

Vous avez découvert le côté `utilisateur` de la POO et à quel point ces nouveaux mécanismes pouvaient vous simplifier la vie.

Le côté `utilisateur` est en fait le côté simple de la POO. Les choses se compliquent lorsqu'on passe du côté `créateur`. Nous allons justement apprendre à créer des objets dans le prochain chapitre et tous les suivants. Une longue route pleine de péripéties nous attend 😊

Les classes (Partie 1/2)

Dans le chapitre précédent, vous avez vu que la programmation orientée objet pouvait nous simplifier la vie en "masquant" en quelque sorte le code complexe. Ca c'est un des avantages de la POO, mais ce n'est pas le seul comme vous allez le découvrir petit à petit : les objets sont aussi facilement réutilisables et modifiables.

A partir de maintenant, **nous allons apprendre à créer des objets**. Vous allez voir que c'est tout un art et que ça demande de la pratique. Il y a beaucoup de programmeurs qui prétendent faire de la POO et qui le font pourtant très mal. En effet, on peut créer un objet de 100 façons différentes, et c'est à nous de choisir à chaque fois la meilleure, la plus adaptée. Pas évident. Il faudra donc bien réfléchir avant de se lancer dans le code comme des forcenés. 🤪

Allez, on prend une **grande** inspiration, et on plonge ensemble dans l'océan de la POO ! 🧐

Créer une classe

Commençons par la question qui doit vous brûler les lèvres. 🤔



Je croyais qu'on allait apprendre à créer des objets, pourquoi tu nous parles de créer une classe maintenant ? Quel est le rapport ?

Eh bien justement, pour créer un objet, il faut d'abord créer une classe !

Je m'explique : pour construire une maison, vous avez besoin d'un plan d'architecte non ? Eh bien imaginez simplement que la classe c'est le plan, et que l'objet c'est la maison.

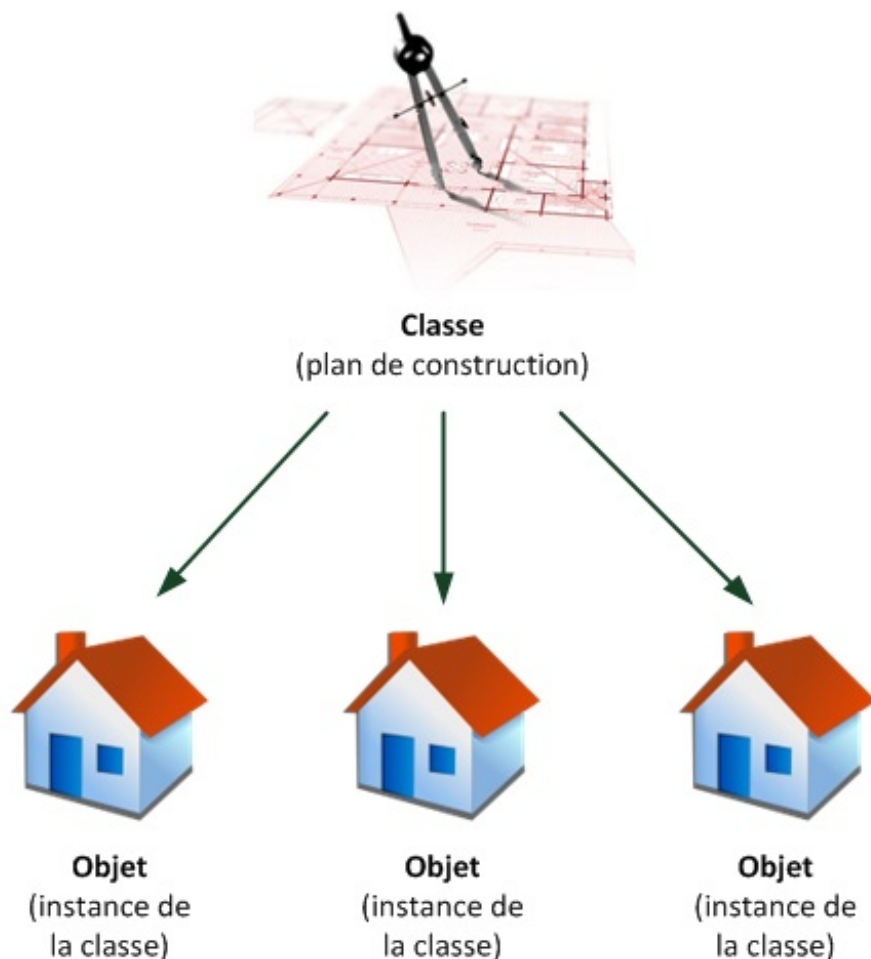
"Créer une classe", c'est donc dessiner les plans de l'objet.

Une fois que vous avez les plans, vous pouvez faire autant de maisons que vous voulez en vous basant sur les plans. Pour les objets c'est pareil : une fois que vous avez fait la classe (le plan), vous pourrez créer autant d'objets du même type que vous voulez. 😊



Vocabulaire : on dit qu'un objet est une **instance** d'une classe. C'est un mot très courant que l'on rencontre souvent en POO. Cela signifie qu'un objet est la matérialisation concrète d'une classe (tout comme la maison est la matérialisation concrète du plan de la maison).

Oui je sais c'est très métaphysique la POO, mais vous allez voir on s'y fait. 😊



Créer une classe, oui mais laquelle ?

Avant tout, il va falloir choisir la classe sur laquelle nous allons travailler.

Pour reprendre mon exemple sur l'architecture : allons-nous créer un appartement, une villa avec piscine, un spacieux loft ? En clair, quel type d'objet voulons-nous être capable de créer ?

Les choix ne manquent pas. Je sais que, quand on débute, on a du mal à imaginer ce qui peut être considéré comme un objet. La réponse est : presque tout !

Vous allez voir, vous allez petit à petit avoir le feeling qu'il faut avec la POO. Puisque vous débutez, c'est moi qui vais choisir (vous avez pas trop le choix de toute façon 🤪).

Pour notre exemple, nous allons créer une classe **Personnage** qui va permettre de représenter un personnage de jeu de rôle (RPG).



Si vous n'avez pas l'habitude des jeux de rôle, rassurez-vous, moi non plus. Vous n'avez pas besoin de savoir jouer à des RPG pour suivre ce chapitre. J'ai choisi cet exemple car il me paraît didactique, amusant, et qu'il peut déboucher sur la création d'un jeu à la fin. Ce sera à vous de le terminer. 😊

Bon, on la crée cette classe ?

C'est parti. 😊

Pour commencer, je vous rappelle qu'une classe est constituée :

- De variables, ici appelées **attributs** (on parle aussi de *variables membres*)

- De fonctions, ici appelées **méthodes** (on parle aussi de *fonctions membres*)

(n'oubliez pas ce vocabulaire, il est fon-da-men-tal !)

Voici le code minimal pour créer une classe :

Code : C++

```
class Personnage
{
    }; // N'oubliez pas le point-virgule à la fin !
```

On utilise comme vous le voyez le mot-clé `class`.

Il est suivi du nom de la classe que l'on veut créer. Ici, c'est `Personnage`.



Souvenez-vous de cette règle très importante : il faut que le nom de vos classes commence toujours par une lettre majuscule ! Bien que ce ne soit pas obligatoire (le compilateur ne hurlera pas si vous commencez par une minuscule), cela vous sera très utile par la suite pour différencier les noms des classes des noms des objets.

Nous allons écrire toute la définition de la classe entre les accolades. Tout ou presque se passera donc à l'intérieur de ces accolades.

Et surtout, super important, le truc qu'on oublie au moins une fois dans sa vie : **il y a un point-virgule après l'accolade fermante**.

Ajout de méthodes et d'attributs

Bon c'est bien beau, mais notre classe `Personnage` est plutôt... vide.

Que va-t-on mettre dans la classe ? Vous le savez déjà voyons. 😊

- Des **attributs**, c'est le nom que l'on donne aux *variables* contenues dans des classes.
- Des **méthodes**, c'est le nom que l'on donne aux *fonctions* contenues dans des classes.

Le but du jeu maintenant, c'est justement d'arriver à faire la liste de tout ce qu'on veut mettre dans notre `Personnage`. De quels attributs et de quelles méthodes a-t-il besoin ? Ca, c'est justement l'étape de *réflexion*, la plus importante. C'est pour ça que je vous ai dit au début de ce chapitre qu'il ne fallait surtout pas coder comme des barbares dès le début, mais prendre le temps de *réfléchir*.



Cette étape de réflexion avant le codage est essentielle quand on fait de la POO. Beaucoup de gens, dont moi, ont l'habitude de sortir une feuille de papier et un crayon pour arriver à établir la liste des attributs et méthodes dont ils vont avoir besoin.

Un langage spécial appelé **UML** a d'ailleurs été spécialement conçu pour "dessiner" les classes avant de commencer à les coder.

Par quoi commencer : les attributs ou les méthodes ? Il n'y a pas d'ordre en fait, mais je trouve un peu plus logique de commencer par voir les attributs *puis* les méthodes.

Les attributs

C'est ce qui va caractériser votre classe, ici le personnage. Ce sont des variables, elles peuvent donc évoluer au fil du temps. Mais qu'est-ce qui caractérise un personnage de jeu de rôle ? Allons, un petit effort. 😊

- Par exemple, tout personnage a un niveau de vie. Hop, ça fait un premier attribut : **vie** ! On dira que ce sera un int, et qu'il sera compris entre 0 et 100 (0 = mort, 100 = toute la vie).
- Dans un jeu de rôle (RPG), il y a le niveau de magie, aussi appelé **mana**. Là encore, on va dire que c'est un int compris entre 0 et 100. Si le personnage a 0 de mana, il ne peut plus lancer de sorts magiques et doit attendre que sa mana se recharge toute seule au fil du temps (ou boire une potion de mana !).
- On pourrait rajouter aussi le nom de l'arme que porte le joueur : **nomArme**. On va utiliser un string pour stocker le nom de l'arme.
- Enfin, il me semble indispensable d'ajouter un attribut **degatsArme**, un int toujours, qui indiquerait cette fois le nombre de dégâts que fait notre arme à chaque coup.

On peut donc déjà commencer à compléter notre classe avec ces premiers attributs :

Code : C++

```
class Personnage
{
    int m_vie;
    int mMana;
    string m_nomArme;
    int m_degatsArme;
};
```

Deux ou trois petites choses à savoir sur ce code :

- Ce n'est pas une obligation, mais une grande partie des programmeurs (dont moi) a l'habitude de faire commencer tous les noms des attributs de classe par **m_** (le "m" signifiant "membre", pour indiquer que c'est une variable membre, c'est-à-dire un attribut). Cela permet de bien différencier les attributs des variables "classiques" (contenues dans des fonctions par exemple).
- Il est impossible d'initialiser les attributs ici. Cela doit être fait via ce qu'on appelle un constructeur, comme on le verra un peu plus loin.
- Comme on utilise un objet string, il faut bien penser à rajouter un `#include <string>` dans votre fichier puisque nous utilisons des chaînes de caractères.

La chose essentielle à retenir ici, est que l'on utilise des attributs pour représenter la notion d'**appartenance**. On dit qu'un Personnage A UNE vie et A UN niveau de magie. Il POSSÈDE également une arme. Lorsque vous repérez une relation d'appartenance, il y a de fortes chances qu'un attribut soit la bonne solution à adopter. 😊

Les méthodes

Les méthodes, elles, sont grosso modo les actions que le personnage peut faire ou qu'on peut lui faire faire. Les méthodes lisent et modifient les attributs.

Voici quelques actions qu'on peut faire avec notre personnage :

- **recevoirDegats** : le personnage prend un certain nombre de dégâts, donc perd de la vie.
- **attaquer** : le personnage attaque un autre personnage avec son arme. Il fait autant de dégâts que son arme lui permet d'en faire (c'est-à-dire degatsArme).
- **boirePotionDeVie** : le personnage boit une potion de vie et regagne un certain nombre de points de vie.
- **changerArme** : le personnage récupère une nouvelle arme plus puissante. On change le nom de l'arme et les dégâts qui vont avec.
- **estVivant** : renvoie vrai si le personnage est toujours vivant (+ que 0 points de vie), renvoie faux sinon.

Voilà c'est un bon début je trouve. 😊

On va rajouter ça dans la classe avant les attributs (on préfère présenter les méthodes *avant* les attributs en POO, bien que ça ne soit pas obligatoire) :

Code : C++

```
class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Attributs
    int m_vie;
    int m_manana;
    string m_nomArme;
    int m_degatsArme;
};
```



Je n'ai pas écrit le code des méthodes exprès, on le fera après. 😊

Ceci dit, vous devriez déjà avoir une petite idée de ce que vous allez mettre dans ces méthodes.

Par exemple, *recevoirDegats* retranchera le nombre de dégâts indiqués en paramètre par *nbDegats* à la vie du personnage. Intéressante aussi : la méthode *attaquer*. Elle prend en paramètre... un autre personnage, plus exactement une référence vers le personnage cible que l'on doit attaquer ! Et que fera cette méthode à votre avis ? Eh oui, elle appellera la méthode *recevoirDegats* de la cible pour lui infliger des dégâts. 😊

Vous commencez à comprendre un peu comment tout cela est lié et terriblement logique ? 😊

On met en général un peu de temps avant de "penser objet" correctement. Si vous vous dites que vous n'auriez pas pu inventer un truc comme ça tout seul, rassurez-vous, tous les débutants passent par là. A force de pratiquer, ça va venir.

Pour info, toutes les méthodes que l'on pourrait créer ne sont pas là : par exemple, on n'utilise pas de magie (mana) ici. Le personnage attaque seulement avec une arme (une épée par exemple) et n'utilise donc pas de sorts magiques. Je laisse exprès quelques fonctions manquantes pour vous inciter à compléter la classe avec vos idées. 😊

En résumé : comme je vous l'avais dit, un objet est bel et bien un mix de "variables" (les attributs) et de "fonctions" (les méthodes). La plupart du temps, les méthodes lisent et modifient les attributs de l'objet pour le faire évoluer. Un objet est au final un petit système intelligent et autonome qui est capable de surveiller son bon fonctionnement tout seul.

Droits d'accès et encapsulation

Nous allons maintenant nous intéresser au concept le plus fondamental de la POO : l'**encapsulation**. Ne vous laissez pas effrayer par ce mot, vous allez vite comprendre ce que ça signifie.

Tout d'abord un petit rappel. En POO, il y a 2 parties bien distinctes :

- On **crée** des classes pour définir le fonctionnement des objets. C'est ce qu'on apprend à faire ici.
- On **utilise** des objets. C'est ce qu'on a appris à faire dans le chapitre précédent.

Il faut bien distinguer ces 2 parties, car ça devient ici très important.

Je mets un exemple création / utilisation côte à côte pour que vous puissiez bien les différencier :

Création de la classe	Utilisation de l'objet
<p>Code : C++</p> <pre>class Personnage { void recevoirDegats (int nbDegats) { } void attaquer (Personnage &cible) { } void boirePotionDeVie (int quantitePotion) { } void changerArme (string nomNouvelleArme, int degatsNouvelleArme) { } bool estVivant() { } int m_vie; int m_manana; string m_nomArme; int m_degatsArme; };</pre>	<p>Code : C++</p> <pre>int main() { Personnage david, goliath; goliath.attaquer(david); david.boirePotionDeVie(20); goliath.attaquer(david); david.attaquer(goliath); goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40); goliath.attaquer(david); return 0; }</pre>

Tenez, pourquoi on n'essaierait pas ce code ?

Allez, on met tout dans un même fichier (en prenant soin de définir la classe *avant* le main), et zou !

Code : C++

```
#include <iostream>
#include <string>

using namespace std;

class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Attributs
    int m_vie;
    int m_manana;
    string m_nomArme;
    int m_degatsArme;
};

int main()
{
    Personnage david, goliath; // Création de 2 objets de type
    Personnage : david et goliath

    goliath.attaquer(david); // goliath attaque david
    david.boirePotionDeVie(20); // david boit une potion de vie qui
    lui rapporte 20 de vie
    goliath.attaquer(david); // goliath réattaque david
    david.attaquer(goliath); // david contre-attaque... c'est assez
    clair non ? ^^
    goliath.changerArme("Double hache tranchante vénéneuse de la
    mort", 40);
    goliath.attaquer(david);

    return 0;
}
```

Compilez et admirez... la belle erreur de compilation ?! 🤔

Error : void Personnage::attaquer(Personnage&) is private within this context

Encore une nouvelle insulte de la part du compilateur !

Les droits d'accès

On en arrive justement au problème qui nous intéresse : celui des droits d'accès (eh ouais j'ai fait exprès de provoquer cette erreur de compilation, vous aviez quand même pas cru que j'avais pas tout prévu ? 🤖).

Ouvrez grand vos oreilles : chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès. Il existe grosso modo 2 droits d'accès différents :

- **public** : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- **private** : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. Par défaut, tous les éléments d'une classe sont private.



Il existe d'autres droits d'accès mais ils sont un peu plus complexes. Nous les verrons plus tard.

Concrètement, qu'est-ce que ça signifie ? Qu'est-ce que "l'extérieur" de l'objet ?

Eh bien sur notre exemple, "l'extérieur" c'est le main. En effet, c'est là où on utilise l'objet. On fait appel à des méthodes, mais comme elles sont privées par défaut, on ne peut pas les appeler depuis le main !

Pour modifier les droits d'accès et mettre par exemple public, il faut taper `public` suivi du symbole `:` (deux points). Tout ce qui se trouvera à la suite sera public.

Voici ce que je vous propose de faire : on va mettre en public toutes les méthodes, et en privé tous les attributs. Ca donne ça :

Code : C++

```
class Personnage
{
    // Tout ce qui suit est public (accessible depuis l'extérieur)
    public:

    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
    private:

    int m_vie;
    int m_manana;
```

```
string m_nomArme;  
int m_degatsArme;  
};
```

Tout ce qui suit le `public` : est public. Donc toutes nos méthodes sont publiques.

Ensuite vient le mot-clé `private` : . Tout ce qui suit ce mot-clé est privé. Donc tous nos attributs sont privés.

Voilà, vous pouvez maintenant compiler ce code, et vous verrez qu'il n'y a pas de problème (même si le code ne fait rien pour l'instant 🤪). On appelle des méthodes depuis le main : comme elles sont publiques, on a le droit de le faire.

... par contre, nos attributs sont privés, ce qui veut dire qu'on n'a pas le droit de les modifier depuis le main. En clair, on ne peut pas écrire dans le main :

Code : C++

```
goliath.m_vie = 90;
```

Essayez, vous verrez que le compilateur vous ressort la même erreur que tout à l'heure : "ton bidule est private... bla bla bla... pas le droit d'appeler un élément private depuis l'extérieur de la classe".

Mais alors... ça veut dire qu'on ne peut pas modifier la vie du personnage depuis le main ? Eh oui ! C'est nul ? Non au contraire, c'est très bien pensé, ça s'appelle l'encapsulation. 🤪

L'encapsulation



Moi j'ai une solution ! Si on mettait tout en public ? Les méthodes ET les attributs en public, comme ça on peut tout modifier depuis le main et plus aucun problème !

... quoi j'ai dit une connerie ? 🤪

Oh, trois fois rien, vous venez juste de vous faire autant d'ennemis qu'il n'y a de programmeurs qui font de la POO dans le monde. 🤪

Il y a une règle d'or en POO, et *tout* découle de là. S'il vous plaît, imprimez ceci en gros sur une feuille, et placez cette feuille sur un mur de votre chambre :

Encapsulation : tous les attributs d'une classe doivent toujours être privés

Ca a l'air bête, stupide, irréfléchi, et pourtant tout ce qui fait que la POO est un principe puissant vient de là.

En clair, si j'en vois un à partir de maintenant qui me met ne serait-ce qu'un seul attribut en public, je le brûle, je le torture, je l'écorche vif sur la place publique, compris ? 🤪

Et vous, si vous voyez quelqu'un d'autre faire ça un jour, écorchez-le vif en pensant à moi, vous serez sympa. 🤪

Voilà qui explique pourquoi j'ai fait exprès dès le début de mettre les attributs en privé. Comme ça, on ne peut pas les modifier depuis l'extérieur de la classe, et ça respecte le principe d'encapsulation.

Vous vous souvenez de ce schéma du chapitre précédent ?



Les fioles chimiques, ce sont les **attributs**.

Les boutons sur la façade avant, ce sont les **méthodes**.

Et là, pif paf pouf, vous devriez avoir tout compris d'un coup. En effet, le but du modèle objet c'est justement de masquer les informations complexes à l'utilisateur (les attributs) pour éviter qu'il ne fasse des bêtises avec.

Imaginez par exemple que l'utilisateur puisse modifier la vie... qu'est-ce qui l'empêcherait de mettre 150 de vie alors que la limite maximale est 100 ? C'est pour ça qu'il faut toujours passer par des méthodes (des fonctions) qui vont *d'abord* vérifier qu'on fait les choses correctement avant de modifier les attributs.

Cela permet de faire en sorte que le contenu de l'objet reste une "boîte noire". On ne sait pas comment ça fonctionne à l'intérieur quand on l'utilise, et c'est très bien. C'est une sécurité, ça permet d'éviter de faire péter tout le bazar de fioles chimiques à l'intérieur. 🤖



Si vous avez fait du C, vous connaissez le mot-clé **struct**. On peut aussi l'utiliser en C++ pour créer des classes. La seule différence avec le mot-clé **class** est que par défaut les méthodes et attributs sont publics au lieu de privés.

Séparer prototypes et définitions

Bon, on avance mais on n'a pas fini !

Voici ce que je voudrais qu'on fasse :

- Séparer les méthodes en prototypes et définitions dans 2 fichiers différents pour avoir un code plus modulaire.
- Implémenter les méthodes de notre classe Personnage (c'est-à-dire écrire le code à l'intérieur parce que pour le moment y'a rien 🤖).

Pour le moment, on a mis notre classe dans le fichier main.cpp, juste au-dessus du main. Et les méthodes sont directement écrites dans la définition de la classe. Ça fonctionne, mais c'est un peu bourrin.

Pour améliorer cela, tout d'abord il faut clairement séparer le main (qui se trouve dans main.cpp) des classes.

Pour *chaque* classe, on va créer :

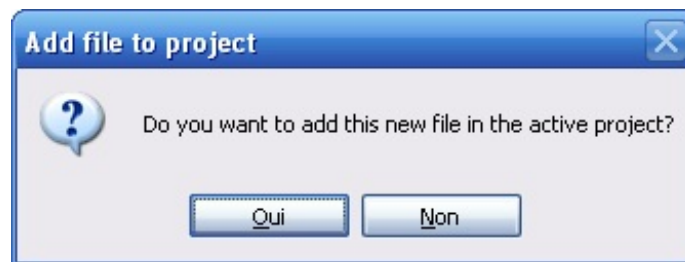
- Un header (fichier *.h) qui contiendra les attributs et les prototypes de la classe
- Un fichier source (fichier *.cpp) qui contiendra la définition des méthodes et leurs implémentations

Je vous propose d'ajouter à votre projet 2 fichiers nommés très exactement :

- Personnage.h
- Personnage.cpp

(vous noterez que je mets aussi une majuscule à la première lettre du nom de fichier, histoire d'être cohérent jusqu'au bout)

Vous devriez être capables de faire ça tous seuls avec votre IDE favori. Sous Code::Blocks, je fais File / New File, je rentre par exemple le nom "Personnage.h" avec l'extension, et je réponds "Oui" quand Code::Blocks me demande si je veux ajouter le nouveau fichier au projet en cours :



Personnage.h

Le fichier .h va donc contenir la déclaration de la classe avec les attributs et les prototypes des méthodes. Dans notre cas, pour la classe Personnage, ça va donner ça :

Code : C++

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include <string>

class Personnage
{
    public:

        void recevoirDegats(int nbDegats);
        void attaquer(Personnage &cible);
        void boirePotionDeVie(int quantitePotion);
        void changerArme(std::string nomNouvelleArme, int
degatsNouvelleArme);
        bool estVivant();

    private:

        int m_vie;
        int m_manana;
        std::string m_nomArme; // Pas de using namespace std, donc il
faut mettre std:: devant string.
        int m_degatsArme;
};

#endif
```

Comme vous pouvez le constater, seuls les prototypes des méthodes sont présents dans le .h. C'est déjà beaucoup plus clair 😊



Dans les .h, il est recommandé de ne jamais mettre la directive using namespace std; car cela pourrait avoir des effets néfastes lorsque vous utiliserez la classe par la suite.



Par conséquent, il faut rajouter le préfixe "std::" devant chaque string du .h. Sinon, le compilateur vous sortira une erreur du type "string does not name a type".

Personnage.cpp

C'est là qu'on va écrire le code de nos méthodes (on dit qu'on **implémente** les méthodes).

La première chose à ne pas oublier, sinon ça va pas bien se passer, c'est d'inclure <string> et "Personnage.h".

On peut aussi rajouter ici un using namespace std;. On a le droit de le faire car on est dans le .cpp (par contre comme je vous l'ai expliqué plus tôt, il faut éviter de le mettre dans le .h).

Code : C++

```
#include "Personnage.h"

using namespace std;
```

Maintenant, voilà comment ça se passe : pour chaque méthode, vous devez faire précéder le nom de la méthode par le nom de la classe suivi de deux fois deux points "::". Pour recevoirDegats ça donne ça :

Code : C++

```
void Personnage::recevoirDegats(int nbDegats)
{
}
}
```

Cela permet au compilateur de savoir que cette méthode se rapporte à la classe Personnage. En effet, comme la méthode est ici écrite en dehors de la définition de la classe, le compilateur n'aurait pas su à quelle classe appartenait cette méthode.

Personnage::recevoirDegats

Maintenant, c'est parti, implémentons la méthode recevoirDegats. Je vous avais expliqué un peu plus haut ce qu'il fallait faire. Vous allez voir, c'est très simple :

Code : C++

```
void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats; // On enlève le nombre de dégâts reçus à la
    vie du personnage

    if (m_vie < 0) // Pour éviter d'avoir une vie négative
    {
        m_vie = 0; // On met la vie à 0 (ça veut dire mort)
    }
}
```

La méthode modifie donc la valeur de la vie. La méthode a le droit de modifier l'attribut, car elle fait partie de la classe. Ne soyez donc pas surpris, c'est justement l'endroit où on a le droit de toucher aux attributs. 😊

La vie est diminuée du nombre de dégâts reçus. En théorie, on aurait pu se contenter de la première instruction, mais on fait une vérification supplémentaire. Si la vie est descendue en-dessous de 0 (parce qu'on a reçu 20 de dégâts alors qu'on n'avait que 10

de vie), on ramène la vie à 0 afin d'éviter d'avoir une vie négative (ça fait pas très pro une vie négative 🤪). De toute façon, à 0 de vie, le personnage est considéré comme mort. 😊

Et voilà pour la première méthode ! Allez on enchaîne hop hop hop !

Personnage::attaquer

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme); // On inflige à la cible
    les dégâts que causent notre arme
}
```

Cette méthode est peut-être très courante, elle n'en est pas moins très intéressante !

On reçoit en paramètre une référence vers un objet de type Personnage. On aurait pu recevoir un pointeur aussi, mais comme les références sont plus faciles à manipuler (cf les chapitres précédents) on ne va pas s'en priver.

La référence concerne le personnage cible que l'on doit attaquer. Pour infliger des dégâts à la cible, on appelle sa méthode recevoirDegats en faisant : cible.recevoirDegats

Quelle quantité de dégâts envoyer à la cible ? Vous avez la réponse sous vos yeux : le nombre de points de dégâts indiqués par l'attribut m_degatsArme ! On envoie donc la valeur des m_degatsArme de notre personnage à la cible.

Personnage::boirePotionDeVie

Code : C++

```
void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) // Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}
```

Le personnage reprend autant de vie que ce que la potion qu'il boit lui permet d'en récupérer. On vérifie au passage qu'il ne dépasse pas les 100 de vie, car comme on l'a dit plus tôt, il est interdit d'avoir plus de 100 de vie.

Personnage::changerArme

Code : C++

```
void Personnage::changerArme(string nomNouvelleArme, int
degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}
```


Pour changer d'arme, on stocke dans nos attributs le nom de la nouvelle arme ainsi que ses nouveaux dégâts. Les instructions sont très simples : on fait juste passer ce qu'on a reçu en paramètres dans nos attributs.

Personnage::estVivant

Code : C++

```
bool Personnage::estVivant()
{
    if (m_vie > 0) // Plus de 0 de vie ?
    {
        return true; // VRAI, il est vivant !
    }
    else
    {
        return false; // FAUX, il n'est plus vivant !
    }
}
```

Cette méthode permet de vérifier si le personnage est toujours vivant. Elle renvoie vrai (true) s'il a plus de 0 de vie, et faux (false) sinon.

Code complet de Personnage.cpp

En résumé, le code complet de notre Personnage.cpp est le suivant :

Code : C++

```
#include "Personnage.h"

using namespace std;

void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats; // On enlève le nombre de dégâts reçus à la
    vie du personnage

    if (m_vie < 0) // Pour éviter d'avoir une vie négative
    {
        m_vie = 0; // On met la vie à 0 (ça veut dire mort)
    }
}

void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme); // On inflige à la cible
    les dégâts que causent notre arme
}

void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) // Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}

void Personnage::changerArme(string nomNouvelleArme, int
```

```

    degatsNouvelleArme)
    {
        m_nomArme = nomNouvelleArme;
        m_degatsArme = degatsNouvelleArme;
    }

    bool Personnage::estVivant()
    {
        if (m_vie > 0) // Plus de 0 de vie ?
        {
            return true; // VRAI, il est vivant !
        }
        else
        {
            return false; // FAUX, il n'est plus vivant !
        }
    }
}

```

main.cpp

Retour au main. Première chose à ne pas oublier : inclure Personnage.h pour pouvoir créer des objets de type Personnage.

Code : C++

```
#include "Personnage.h" // Ne pas oublier
```

Après, le main reste le même que tout à l'heure, on n'a pas besoin de le changer. Au final, le code du main est donc très court, et le fichier main.cpp ne fait qu'**utiliser** les objets :

Code : C++

```

#include <iostream>
#include "Personnage.h" // Ne pas oublier

using namespace std;

int main()
{
    Personnage david, goliath; // Création de 2 objets de type
    Personnage : david et goliath

    goliath.attaquer(david); // goliath attaque david
    david.boirePotionDeVie(20); // david boit une potion de vie qui
    lui rapporte 20 de vie
    goliath.attaquer(david); // goliath réattaque david
    david.attaquer(goliath); // david contre-attaque... c'est assez
    clair non ? ^^
    goliath.changerArme("Double hache tranchante vénéneuse de la
    mort", 40);
    goliath.attaquer(david);

    return 0;
}

```

N'exécutez pas le programme pour le moment. En effet, nous n'avons toujours pas vu comment faire pour initialiser les



attributs, ce qui fait que notre programme n'est pas encore utilisable.

Nous verrons comment le rendre pleinement fonctionnel dans le chapitre suivant, et vous pourrez alors (enfin) l'exécuter. 😊

Il faudra donc pour le moment vous contenter de votre imagination. Essayez d'imaginer que David et Goliath sont bien en train de combattre ! (et je veux pas faire mon gros spoiler, mais normalement c'est David qui gagne à la fin 🤔).

Là, on peut dire qu'on est rentré en plein dans la POO. 😊

Pourtant, ce n'est encore qu'un début ! De nombreuses nouvelles choses complètement dingues vous attendent dans les chapitres qui suivent (et elles vont vous rendre dingues ça c'est sûr 🤔).

Un conseil si je puis me permettre : assurez-vous d'avoir bien compris qu'il y avait deux faces dans la POO, la création de la classe, et l'utilisation des objets. Il faut être à l'aise avec ce concept.

Mais tout n'est pas si simple. Comme vous le verrez, ce que font les objets la plupart du temps c'est... utiliser d'autres objets ! Et c'est en combinant plusieurs objets entre eux que l'on découvrira le vrai pouvoir de la POO 😊

Les classes (Partie 2/2)

Allez, hop hop hop, on enchaîne ! Pas question de s'endormir, on est en plein dans la POO là. 😊

Dans le chapitre précédent, nous avons appris à créer une classe basique, à rendre le code modulaire en POO, et surtout nous avons découvert le principe d'encapsulation (suuuper important l'encapsulation, c'est la base de tout je le rappelle).

Dans cette seconde partie du chapitre, nous allons découvrir comment initialiser nos attributs à l'aide d'un constructeur, un élément indispensable à toute classe qui se respecte. Puisqu'on parlera de constructeur, on parlera aussi de destructeur, ça va de pair vous verrez.

Nous compléterons notre classe `Personnage` et nous l'associerons avec une nouvelle classe `Arme` que nous allons créer. Nous découvrirons alors tout le pouvoir qu'il y a de combiner des classes entre elles, et vous devriez normalement commencer à imaginer pas mal de possibilités à partir de là. 😊

Constructeur et destructeur

Reprenons. Nous avons maintenant 3 fichiers :

- **main.cpp** : il contient le main, dans lequel on a créé 2 objets de type `Personnage` : david et goliath.
- **Personnage.h** : c'est le header de la classe `Personnage`. On y liste les prototypes des méthodes et les attributs. On y définit la portée (public / private) de chacun des éléments. Pour respecter le principe d'encapsulation, tous nos attributs sont privés, c'est-à-dire non accessibles de l'extérieur.
- **Personnage.cpp** : c'est le fichier dans lequel on implémente nos méthodes, c'est-à-dire qu'on écrit le code source des méthodes.

Pour l'instant, nous avons défini et implémenté pas mal de méthodes. Je voudrais vous parler ici de 2 méthodes particulières que l'on retrouve dans la plupart des classes : le constructeur et le destructeur.

- **Le constructeur** : c'est une méthode qui est appelée automatiquement à chaque fois que l'on crée un objet basé sur cette classe.
- **Le destructeur** : c'est une méthode qui est automatiquement appelée lorsqu'un objet est détruit, par exemple à la fin de la fonction dans laquelle il a été déclaré ou lors d'un delete si l'objet a été alloué dynamiquement avec new.

Voyons voir plus en détail comment fonctionnent ces méthodes un peu particulières...

Le constructeur

Comme son nom l'indique, c'est une méthode qui sert à *construire* l'objet. Dès qu'on crée un objet, le constructeur est automatiquement appelé.

Par exemple, lorsqu'on fait dans notre main :

Code : C++

```
Personnage david, goliath;
```

Le constructeur de l'objet david est appelé, et de même pour le constructeur de l'objet goliath.



Un constructeur par défaut est automatiquement créé par le compilateur. C'est un constructeur vide, qui ne fait rien de particulier.

On a cependant très souvent besoin de créer nous-mêmes un constructeur, qui remplace ce constructeur vide par défaut.

Le rôle du constructeur

Si le constructeur est appelé lors de la création de l'objet, ce n'est pas pour faire joli. En fait, le rôle principal du constructeur est d'*initialiser* les attributs.

En effet, souvenez-vous : nos attributs sont déclarés dans `Personnage.h`, mais pas initialisés !

Revoici `Personnage.h` :

Code : C++

```
#include <string>

class Personnage
{
    public:

        void recevoirDegats(int nbDegats);
        void attaquer(Personnage &cible);
        void boirePotionDeVie(int quantitePotion);
        void changerArme(std::string nomNouvelleArme, int
degatsNouvelleArme);
        bool estVivant();

    private:

        int m_vie;
        int m_mana;
        std::string m_nomArme;
        int m_degatsArme;
};
```

Nos attributs `m_vie`, `m_mana`, et `m_degatsArmes` ne sont pas initialisés ! Pourquoi ? Parce qu'on n'a pas le droit d'initialiser les attributs ici. C'est justement dans le constructeur qu'il faut le faire.



En fait, le constructeur est indispensable pour initialiser les attributs qui ne sont pas des objets (type classique : `int`, `double`, `char...`). En effet, ceux-ci ont une valeur inconnue en mémoire (ça peut être 0 comme -3451). En revanche, les attributs qui sont des objets, comme c'est le cas de `m_nomArme` ici qui est un `string`, sont automatiquement initialisés par le langage C++ avec une valeur par défaut.

Créer un constructeur

Le constructeur est une méthode, mais une méthode un peu particulière.

En effet, pour créer un constructeur, il y a 2 règles à respecter :

- Il faut que la méthode ait le même nom que la classe. Dans notre cas, la méthode devra s'appeler "Personnage".
- La méthode ne doit RIEN renvoyer, pas même `void` ! C'est une méthode sans aucun type de retour.

Si on déclare son prototype dans `Personnage.h`, ça donne ça :

Code : C++

```
#include <string>

class Personnage
{
    public:

    Personnage(); // Constructeur
```

```

    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int
degatsNouvelleArme);
    bool estVivant();

private:
    int m_vie;
    int m_manana;
    std::string m_nomArme;
    int m_degatsArme;
};

```

Le constructeur se voit du premier coup d'oeil : déjà parce qu'il n'a aucun type de retour (pas de void ni rien), et ensuite parce qu'il a le même nom que la classe. 😊

Et si on en profitait pour implémenter ce constructeur dans Personnage.cpp maintenant ? 😊

Voici à quoi pourrait ressembler son implémentation :

Code : C++

```

Personnage::Personnage()
{
    m_vie = 100;
    m_manana = 100;
    m_nomArme = "Epée rouillée";
    m_degatsArme = 10;
}

```

Vous noterez une fois de plus qu'il n'y a pas de type de retour, pas même void (très important, c'est une erreur que l'on fait souvent 😊).

J'ai choisi de mettre la vie et la mana à 100, le maximum, ce qui est logique. J'ai mis par défaut une arme appelée "Epée rouillée" qui fait 10 de dégâts à chaque coup.

Et voilà ! Notre classe Personnage a un constructeur qui initialise les attributs, elle est désormais pleinement utilisable. 😊

Maintenant, à chaque fois que l'on crée un objet de type Personnage, celui-ci est initialisé à 100 points de vie et de mana, avec l'arme "Epée rouillée". Nos deux compères david et goliath commencent donc à égalité lorsqu'ils sont créés dans le main :

Code : C++

```

Personnage david, goliath; // Les constructeurs de david et goliath
sont appelés.

```

Autre façon d'initialiser avec un constructeur : la liste d'initialisation

Le C++ permet d'initialiser les attributs de la classe d'une autre manière (un peu déroutante) appelée **liste d'initialisation**. C'est une technique que je vous recommande d'utiliser quand vous le pouvez (et c'est celle que nous utiliserons dans ce cours).

Reprenons le constructeur qu'on vient de créer :

Code : C++

```

Personnage::Personnage()

```

```
{  
    m_vie = 100;  
    m_mana = 100;  
    m_nomArme = "Epée rouillée";  
    m_degatsArme = 10;  
}
```

Le code que vous allez voir ci-dessous produit le même effet :

Code : C++

```
Personnage::Personnage() : m_vie(100), m_mana(100), m_nomArme("Epée  
rouillée"), m_degatsArme(10)  
{  
    // Rien à mettre dans le corps du constructeur, tout a déjà été  
    fait !  
}
```

La nouveauté, c'est qu'on rajoute un symbole deux-points (:) suivi de la liste des attributs que l'on veut initialiser avec la valeur entre parenthèses. Avec ce code, on initialise la vie à 100, la mana à 100, l'attribut m_nomArme à "Epée rouillée", etc.

Cette technique est un peu surprenante, surtout que du coup on n'a plus rien à mettre dans le corps du constructeur entre les accolades, vu que tout a déjà été fait avant ! Elle a toutefois l'avantage d'être "plus propre" et se révélera pratique dans la suite du chapitre.

On va donc utiliser autant que possible les listes d'initialisation avec les constructeurs, c'est une bonne habitude à prendre.



Le prototype du constructeur (dans le .h) ne change pas. Toute la partie après les deux-points n'apparaît pas dans le prototype.

Surcharger le constructeur

Vous savez qu'en C++ on a le droit de surcharger les fonctions, donc de surcharger les méthodes. Et comme le constructeur est une méthode, on a le droit de le surcharger lui aussi.

Pourquoi je vous en parle ? Ce n'est pas par hasard : en fait, le constructeur est une méthode que l'on a tendance à beaucoup surcharger. Cela permet de créer un objet de plusieurs façons différentes.

Pour l'instant, on a créé un constructeur sans paramètres :

Code : C++

```
Personnage();
```

On appelle ça : **le constructeur par défaut** (il fallait bien lui donner un nom le pauvre 🤔).

Supposons que l'on souhaite créer un personnage qui ait dès le départ une meilleure arme... comment diable faire ?

C'est là que la surcharge devient utile. On va créer un 2ème constructeur qui prendra en paramètre le nom de l'arme et ses dégâts.

Dans Personnage.h, on va donc rajouter ce prototype :

Code : C++

```
Personnage(std::string nomArme, int degatsArme);
```



Le préfixe `std::` est obligatoire ici comme je vous l'ai dit plus tôt car on n'utilise pas la directive `using namespace std;` dans le `.h` (cf chapitre précédent).

L'implémentation dans `Personnage.cpp` sera la suivante :

Code : C++

```
Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100),
mMana(100), m_nomArme(nomArme), m_degatsArme(degatsArme)
{
}
}
```

Vous noterez ici tout l'intérêt de mettre le préfixe `m_` au début des attributs : comme ça on peut faire la différence dans notre code entre `m_nomArme`, qui est un attribut, et `nomArme`, qui est le paramètre envoyé au constructeur.

Ici, on place juste dans l'attribut de l'objet le nom de l'arme envoyé en paramètre. On recopie juste la valeur. C'est tout bête, mais il faut le faire, sinon l'objet ne se "souviendra pas" du nom de l'arme qu'il possède.

La vie et la mana, eux, sont toujours fixés à 100 (il faut bien les initialiser), mais l'arme, elle, peut maintenant être indiquée par l'utilisateur lorsqu'il crée l'objet.



Quel utilisateur ? 🤔

Souvenez-vous, l'utilisateur c'est celui qui crée et utilise les objets. Le concepteur c'est celui qui crée les classes.

Dans notre cas, la création des objets est faite dans le `main`. Pour le moment, la création de nos objets ressemble à ça :

Code : C++

```
Personnage david, goliath;
```

Comme on n'a spécifié aucun paramètre, c'est le constructeur par défaut (celui sans paramètres) qui sera appelé.

Maintenant supposons que l'on veuille donner dès le départ une meilleure arme à Goliath (c'est lui le plus fort après tout 🤖).

On va indiquer entre parenthèses le nom et la puissance de cette arme :

Code : C++

```
Personnage david, goliath("Epée aiguisée", 20);
```

Goliath est équipé de l'épée aiguisée dès sa création. David est équipé de l'arme par défaut, l'épée rouillée.

Comme on n'a spécifié aucun paramètre lors de la création de `david`, c'est le constructeur par défaut qui sera appelé pour lui. Pour `goliath`, comme on a spécifié des paramètres, c'est le constructeur qui prend en paramètre un `string` et un `int` qui sera appelé.

Exercice : on aurait aussi pu permettre à l'utilisateur de modifier la vie et la mana de départ, mais je ne l'ai pas fait ici. Ce n'est pas compliqué, vous pouvez le faire pour vous entraîner. Ça vous fera un troisième constructeur surchargé. 😊

Le destructeur

Le destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (via des delete) qui a été allouée dynamiquement.

Dans le cas de notre classe Personnage, on n'a fait aucune allocation dynamique (il n'y a aucun new). Le destructeur est donc inutile. Cependant, vous en aurez certainement besoin un jour où l'autre, car on est souvent amené à faire des allocations dynamiques.

Tenez, l'objet string par exemple, vous croyez qu'il fonctionne comment ? Il a un destructeur qui lui permet, juste avant la destruction de l'objet, de supprimer le tableau de char qu'il a alloué dynamiquement en mémoire. Il fait donc un delete sur le tableau de char, ce qui permet de garder une mémoire propre et d'éviter les fameuses "fuites de mémoire". 🤔

Créer un destructeur

Bien que ce soit inutile dans notre cas (je n'ai pas mis d'allocations dynamiques pour ne pas trop compliquer de suite 😊), je vais vous montrer comment on crée un destructeur. Voici les règles à suivre :

- Un destructeur est une méthode qui commence par un tilde ~ suivi du nom de la classe
- Un destructeur ne renvoie aucune valeur, pas même void (comme le constructeur)
- Et, nouveauté : le destructeur ne peut prendre aucun paramètre. Il y a donc toujours un seul destructeur, il ne peut pas être surchargé.

Dans Personnage.h, le prototype du destructeur sera donc :

Code : C++

```
~Personnage () ;
```

Dans Personnage.cpp, l'implémentation sera :

Code : C++

```
Personnage::~~Personnage ()
{
    /* Rien à mettre ici car on ne fait pas d'allocation dynamique
    dans la classe Personnage. Le destructeur est donc inutile mais
    je le mets pour montrer à quoi ça ressemble.
    En temps normal, un destructeur fait souvent des delete et quelques
    autres vérifications si nécessaire avant la destruction de l'objet
    */
}
```

Bon vous l'aurez compris, mon destructeur ne fait rien. C'était même pas la peine de le créer (il n'est pas obligatoire après tout). Cela vous montre néanmoins la procédure à suivre. Soyez rassurés, nous ferons des allocations dynamiques plus tôt que vous ne le pensez (je sais je suis diabolique 😈), et nous aurons alors grand besoin du destructeur pour désallouer la mémoire !

Les méthodes constantes

Les méthodes constantes sont des méthodes de "lecture seule". Elles possèdent le mot-clé **const** à la fin de leur prototype et de leur déclaration.

Quand vous dites "ma méthode est constante", vous indiquez au compilateur que votre méthode ne modifie pas l'objet, c'est-à-dire qu'elle ne modifie la valeur d'aucun de ses attributs. Par exemple, une méthode qui se contente d'afficher des informations à l'écran sur l'objet est une méthode constante : elle ne fait que lire les attributs. En revanche, une méthode qui met à jour le niveau de vie d'un personnage ne peut pas être constante 😊

Ça s'utilise comme ceci :

Code : C++

```
// Prototype de la méthode (dans le .h) :  
void maMethode(int parametre) const;  
  
// Déclaration de la méthode (dans le .cpp) :  
void MaClasse::maMethode(int parametre) const  
{  
  
}
```

On utilisera le mot-clé **const** sur des méthodes qui se contentent de renvoyer des informations sans modifier l'objet. C'est le cas par exemple de la méthode `estVivant()` qui indique si le Personnage est toujours vivant ou non. Elle ne modifie pas l'objet, elle se contente juste de vérifier le niveau de vie.

Code : C++

```
bool Personnage::estVivant() const  
{  
    if (m_vie > 0)  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```



En revanche, une méthode comme `recevoirDegats()` ne peut pas être déclarée constante ! En effet, elle modifie le niveau de vie du Personnage puisque celui-ci reçoit des dégâts.

On pourrait trouver d'autres exemples de méthodes qui seraient concernées. Pensez par exemple à la méthode `size()` de la classe `string`. Elle ne modifie pas l'objet, elle ne fait que nous informer de la longueur du texte contenu dans la chaîne.



Concrètement, ça sert à quoi de créer des méthodes constantes ?

Ca sert à 3 choses principalement :

- **Pour vous** : vous savez que votre méthode ne fait que lire les attributs, et vous vous interdisez dès le début de les modifier. Si par erreur vous en modifiez, le compilateur plantera en vous disant que vous ne respectez pas la règle que vous vous êtes fixée. Et ça c'est bien.
- **Pour les utilisateurs de votre classe** : c'est très important aussi pour eux, ça leur indique que la méthode ne fait que renvoyer un résultat mais qu'elle ne modifie pas l'objet. Dans une documentation, le mot-clé **const** apparaît dans le prototype de la méthode et est un excellent indicateur de ce qu'elle fait, ou plutôt de ce qu'elle ne peut pas faire (ça pourrait se traduire par : "*cette méthode ne modifiera pas votre objet*").
- **Pour le compilateur** : si vous vous rappelez du chapitre sur les variables, je vous conseillai de toujours déclarer **const** ce qui peut l'être. On est dans le même cas ici. On offre des garanties aux utilisateurs de la classe et on aide le compilateur à générer du meilleur code binaire.

Associer des classes entre elles

La programmation orientée objet devient vraiment intéressante et puissante lorsqu'on se met à combiner plusieurs objets entre eux. Pour l'instant, nous n'avons créé qu'une seule classe : `Personnage`.

Or en pratique, un programme objet est un programme constitué d'une multitude d'objets différents !

Il n'y a pas de secret, c'est en pratiquant que l'on apprend petit à petit à penser objet.

Ce que nous allons voir par la suite ne sera pas nouveau : vous allez réutiliser tout ce que vous savez déjà sur la création de classes, de manière à améliorer notre petit RPG et à vous entraîner encore plus à manipuler des objets. 😊

La classe Arme

Je vous propose dans un premier temps de créer une nouvelle classe Arme. Plutôt que de mettre les informations de l'arme (m_nomArme, m_degatsArme) directement dans le Personnage, **nous allons l'équiper d'un objet de type Arme**. Le découpage de notre programme sera alors un peu plus dans la logique d'un programme orienté objet.



Souvenez-vous ce que je vous ai dit au début : il y a 100 façons différentes de concevoir un même programme en POO. Tout est dans l'organisation des classes entre elles, comment elles communiquent, etc. Ce que nous avons fait jusqu'ici était pas mal, mais je veux vous montrer ici qu'on peut faire *autrement*, un peu plus dans l'esprit objet, donc... mieux. 😊

Qui dit nouvelle classe dit 2 nouveaux fichiers :

- Arme.h : contient la définition de la classe
- Arme.cpp : contient l'implémentation des méthodes de la classe



On n'est pas obligé de procéder ainsi. On pourrait tout mettre dans un seul fichier. On pourrait même mettre plusieurs classes par fichier, rien ne l'interdit en C++. Cependant, pour des raisons d'organisation, je vous recommande de faire comme moi.

Arme.h

Voici ce que je propose de mettre dans Arme.h :

Code : C++

```
#ifndef DEF_ARME
#define DEF_ARME

#include <iostream>
#include <string>

class Arme
{
public:
    Arme();
    Arme(std::string nom, int degats);
    void changer(std::string nom, int degats);
    void afficher() const;

private:
    std::string m_nom;
    int m_degats;
};

#endif
```

Mis à part les includes qu'il ne faut pas oublier, le reste de la classe est très simple.

On met le nom de l'arme et ses dégâts dans des attributs, et comme ce sont des attributs, on vérifie qu'ils soient bien privés (encapsulation). Vous remarquerez qu'au lieu de `m_nomArme` et `m_degatsArme`, j'ai choisi de nommer mes attributs `m_nom` et `m_degats` tout simplement. C'est plus logique en effet : vu qu'on est *déjà* dans l'Arme, ce n'est pas la peine de préciser dans les attributs qu'il s'agit de l'arme, on le sait déjà, on est dedans. 😊

Ensuite, on ajoute un ou deux constructeurs, une méthode pour changer d'arme à tout moment, et une autre allez, soyons fous 🤪, pour afficher le contenu de l'arme.

Reste à implémenter toutes ces méthodes dans `Arme.cpp`. Pfeuh, fastoche ! 😊

Arme.cpp

Entraînez-vous à écrire `Arme.cpp`, c'est tout bête, les méthodes font maxi 2 lignes, bref c'est à la portée de tout le monde. 😊

Voici mon `Arme.cpp` pour comparer :

Code : C++

```
#include "Arme.h"

using namespace std;

Arme::Arme() : m_nom("Epée rouillée"), m_degats(10)
{
}

Arme::Arme(string nom, int degats) : m_nom(nom), m_degats(degats)
{
}

void Arme::changer(string nom, int degats)
{
    m_nom = nom;
    m_degats = degats;
}

void Arme::afficher() const
{
    cout << "Arme : " << m_nom << " (Dégâts : " << m_degats << " ) "
    << endl;
}
```

N'oubliez pas d'inclure `"Arme.h"` si vous voulez que ça marche. 😊

Et ensuite ?

Bon, notre classe `Arme` est créée, c'est bon pour ça. Mais maintenant, il va falloir adapter la classe `Personnage` pour qu'elle utilise non pas `m_nomArme` et `m_degatsArme`, mais un objet de type `Arme`. Et là... c'est là que ça se complique. 😊

Adapter la classe Personnage pour utiliser une Arme

La classe `Personnage` va subir quelques modifications pour utiliser la classe `Arme`. Restez attentifs, car utiliser un objet **DANS** un objet, c'est un peu particulier.

Personnage.h

Zou, direction le .h. On commence par virer nos 2 attributs `m_nomArme` et `m_degatsArme` qui ne servent plus à rien.

Les méthodes n'ont pas besoin d'être changées. En fait, il ne vaut mieux pas les changer. Pourquoi ? Parce que les méthodes sont déjà potentiellement utilisées par quelqu'un (par exemple dans notre main). Si on les renomme ou si on en supprime, notre programme ne fonctionnera plus.

Ce n'est peut-être pas grave pour un si petit programme, mais dans le cas d'un gros programme si on supprime une méthode, c'est la cata assurée dans le reste du programme. Et je vous parle même pas de ceux qui écrivent des bibliothèques C++ : si d'une version à l'autre des méthodes disparaissent, tous les programmes qui utilisent la librairie ne fonctionneront plus ! 💡

Je vais peut-être vous surprendre en vous disant ça, mais c'est là tout l'intérêt de la programmation orientée objet, et plus particulièrement de l'**encapsulation**. On peut changer nos attributs comme on veut, vu qu'ils ne sont pas accessibles de l'extérieur, on ne prend pas le *risque* que quelqu'un les utilise déjà dans le programme.

En revanche, pour les méthodes, faites plus attention. Vous pouvez ajouter de nouvelles méthodes, modifier l'implémentation des méthodes existantes, mais PAS en supprimer ou en renommer, sinon l'utilisateur risque d'avoir des problèmes.

Cette petite réflexion sur l'encapsulation étant faite (vous en comprendrez tout le sens avec la pratique 😊), il va falloir ajouter un objet de type `Arme` à notre `Personnage`.



Il faut penser à ajouter un `include` de `"Arme.h"` si on veut pouvoir utiliser un objet de type `Arme`.

Voici mon nouveau `Personnage.h` :

Code : C++

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include <iostream>
#include <string>
#include "Arme.h" // Ne PAS oublier d'inclure Arme.h pour en avoir la définition

class Personnage
{
public:
    Personnage();
    Personnage(std::string nomArme, int degatsArme);
    ~Personnage();
    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant() const;

private:
    int m_vie;
    int m_mana;
    Arme m_arme; // Notre Personnage possède une Arme
};

#endif
```

Personnage.cpp

Nous n'avons besoin de changer que les méthodes qui utilisent l'arme pour les adapter.
On commence par les constructeurs :

Code : C++

```
Personnage::Personnage() : m_vie(100), m_mana(100)
{

}

Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100),
m_mana(100), m_arme(nomArme, degatsArme)
{

}
```

Notre objet `m_arme` est ici initialisé avec les valeurs reçues en paramètre par `Personnage(nomArme, degatsArme)`. C'est là que la liste d'initialisation devient utile. En effet, on n'aurait pas pu initialiser `m_arme` sans une liste d'initialisation !

Peut-être ne voyez-vous pas bien pourquoi. Conseil perso : ne vous prenez pas la tête à essayer de comprendre le pourquoi du comment ici, et contentez-vous de toujours utiliser les listes d'initialisation avec vos constructeurs, ça vous évitera bien des problèmes.

Revenons au code.

Dans le premier constructeur, c'est le constructeur par défaut de la classe `Arme` qui est appelé, tandis que dans le second c'est celui qui prend en paramètre un `string` et un `int` qui est appelé.

La méthode `recevoirDegats` n'a pas besoin de changer.

En revanche, la méthode `attaquer` est délicate. En effet, on ne peut pas faire :

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.m_degats);
}
```

Pourquoi est-ce interdit ? Parce que `m_degats` est un attribut, et que comme tout bon attribut qui se respecte, il est *privé* ! Diantre... On est en train d'utiliser la classe `Arme` au sein de la classe `Personnage`, et comme on est utilisateurs, on ne peut pas accéder aux éléments privés. 🙄

(La POO, ça peut parfois donner mal à la tête j'avais oublié de vous prévenir 🤪)

Bon, comment résoudre le problème ? Il n'y a pas 36 solutions. Ça va peut-être vous surprendre, mais on doit créer une méthode pour récupérer la valeur de cet attribut. Cette méthode est appelée *accesseur* et commence généralement par le préfixe `get` (*récupérer*, en anglais). Dans notre cas, notre méthode s'appellerait `getDegats`.

On conseille généralement de rajouter le mot-clé `const` aux accesseurs pour en faire des méthodes constantes, puisqu'elles ne modifient pas l'objet.

Code : C++

```
int Arme::getDegats() const
{
    return m_degats;
}
```

N'oubliez pas de mettre à jour Arme.h avec le prototype aussi, qui sera le suivant :

Code : C++

```
int getDegats() const;
```

Voilà, ça peut paraître idiot, et pourtant c'est une sécurité nécessaire. On est parfois obligé de créer une méthode qui fait juste un return pour accéder indirectement à un attribut.



De même, on crée parfois des accesseurs permettant de modifier des attributs. Ces accesseurs sont généralement précédés du préfixe *set* (*mettre*, en anglais).

Vous avez peut-être l'impression qu'on viole la règle d'encapsulation ? Eh bien non. Car la méthode nous permet de faire des tests pour vérifier qu'on ne met pas n'importe quoi dans l'attribut, donc ça reste une façon sécurisée de modifier un attribut.

Vous pouvez maintenant retourner dans Personnage.cpp et écrire :

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.getDegats());
}
```

getDegats renvoie le nombre de dégâts, qu'on envoie à la méthode recevoirDegats de la cible. Pfiou ! 😊

Le reste des méthodes n'a pas besoin de changer, à part changerArme de la classe Personnage :

Code : C++

```
void Personnage::changerArme(string nomNouvelleArme, int
degatsNouvelleArme)
{
    m_arme.changer(nomNouvelleArme, degatsNouvelleArme);
}
```

On appelle la méthode changer de m_arme.

Le Personnage répercute donc la demande de changement d'arme à la méthode changer de son objet m_arme.

Comme vous pouvez le voir, on peut faire communiquer des objets entre eux, à condition d'être bien organisé et de se demander à chaque instant "est-ce que j'ai le droit d'accéder à cet élément ou pas ?".

N'hésitez pas à créer des accesseurs si besoin est, même si ça peut paraître lourd c'est la bonne méthode. En aucun cas vous ne devez mettre un attribut public pour simplifier un problème. Vous perdriez tous les avantages et la sécurité de la POO (et vous n'auriez aucun intérêt à continuer le C++ dans ce cas 🤪).

Action !

Nos personnages combattent dans le main, mais... on ne voit rien de ce qui se passe. Il serait bien d'afficher l'état de chacun des personnages pour savoir où ils en sont.

Je vous propose de créer une méthode `afficherEtat` dans `Personnage`. Cette méthode sera chargée de faire des *cout* pour afficher dans la console la vie, la mana et l'arme du personnage.

Prototype et include

On va rajouter le prototype, tout bête, dans le `.h` :

Code : C++

```
void afficherEtat() const;
```

Implémentation

Implémentons ensuite la méthode. C'est simple, on a juste des `cout` à faire. Grâce aux attributs, on peut indiquer toutes les infos sur le personnage :

Code : C++

```
void Personnage::afficherEtat() const
{
    cout << "Vie : " << m_vie << endl;
    cout << "Mana : " << m_mana << endl;
    m_arme.afficher();
}
```

Comme vous pouvez le voir, les informations sur l'arme sont demandées à l'objet `m_arme` via sa méthode `afficher()`. Encore une fois, les objets communiquent entre eux pour récupérer les informations dont ils ont besoin.

Appel de `afficherEtat` dans le main

Bien, tout ça c'est bien beau, mais tant qu'on n'appelle pas la méthode, elle ne sert à rien 🤪

Je vous propose donc de compléter le main et de rajouter à la fin les appels de méthode :

Code : C++

```
int main()
{
    // Création des personnages
    Personnage david, goliath("Epée aiguisée", 20);

    // Au combat !
    goliath.attaquer(david);
    david.boirePotionDeVie(20);
    goliath.attaquer(david);
    david.attaquer(goliath);
    goliath.changerArme("Double hache tranchante vénéneuse de la
mort", 40);
    goliath.attaquer(david);

    // Temps mort ! Voyons voir la vie de chacun...
    cout << "David" << endl;
```



```
david.afficherEtat();  
cout << endl << "Goliath" << endl;  
goliath.afficherEtat();  
  
return 0;  
}
```

On peut *enfin* exécuter le programme et voir quelque chose dans la console 😊

Code : Console

```
David  
Vie : 40  
Mana : 100  
Arme : Epée rouillée (Degats : 10)  
  
Goliath  
Vie : 90  
Mana : 100  
Arme : Double hache tranchante vénéneuse de la mort (Degats : 40)
```



Si vous êtes sous Windows, vous aurez probablement un bug avec les accents dans la console. Ignorez-le, ne vous en préoccupez pas, ce qui nous intéresse c'est le fonctionnement de la POO ici. Et puis de toute manière, dans la prochaine partie du cours on travaillera avec de vraies fenêtres, donc la console c'est temporaire pour nous. 🤔

Pour que vous puissiez vous faire une bonne idée du projet dans son ensemble, je vous propose de télécharger un fichier zip contenant :

- main.cpp
- Personnage.cpp
- Personnage.h
- Arme.cpp
- Arme.h

... bref, c'est-à-dire tout le projet tel qu'il est sur mon ordinateur à l'heure actuelle.

Télécharger le projet RPG (3 Ko)

Je vous invite à faire des tests pour vous entraîner. Par exemple :

- Continuez à faire combattre david et goliath dans le main en affichant leur état de temps en temps.
- Introduisez un troisième personnage dans l'arène pour rendre le combat plus ~~brutal~~ intéressant. 🤔
- Rajoutez un attribut m_nom pour stocker le nom du personnage dans l'objet. Pour le moment, nos personnages ne savent même pas comment ils s'appellent, c'est un peu bête. 😊
Du coup, je pense qu'il faudrait modifier les constructeurs et obliger l'utilisateur à indiquer un nom pour le personnage lors de sa création... à moins que vous ne donniez un nom par défaut si rien n'est précisé ? A vous de choisir !
- Rajoutez des cout dans les autres méthodes de Personnage pour indiquer à chaque fois ce qui est en train de se passer ("machin boit une potion qui lui redonne 20 points de vie")
- Rajoutez d'autres méthodes au gré de votre imagination... et pourquoi pas des attaques magiques qui utilisent de la mana ?
- Enfin, pour l'instant le combat est écrit dans le main, mais vous pourriez laisser le joueur choisir les attaques dans la console à l'aide de cin. Vous savez le faire, allez allez !

Prenez cet exercice très au sérieux, ceci est peut-être la base de votre futur MMORPG révolutionnaire !

Précision utile : la phrase ci-dessus était une boutade. 🤪



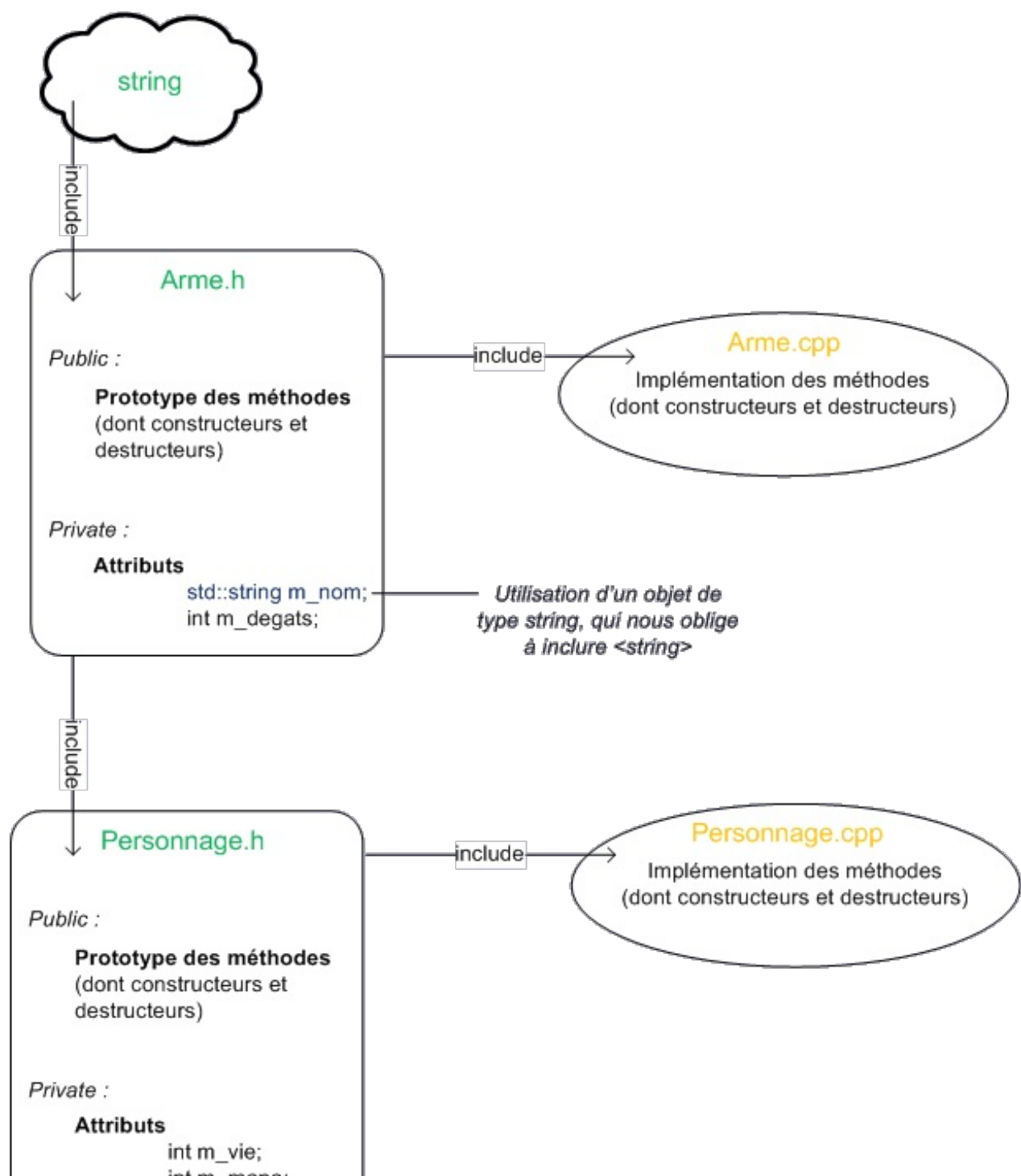
Ce cours ne vous apprendra pas à créer un MMORPG, vu le travail phénoménal que cela représente. Mieux vaut commencer par se concentrer sur de plus petits projets réalistes, et notre RPG en est un. Ce qui est intéressant ici, c'est de voir comment est conçu un jeu orienté objet (comme c'est le cas de la plupart des jeux aujourd'hui). Si vous avez bien compris le principe, vous devriez commencer à voir des objets dans tous les jeux que vous connaissez ! Par exemple, un bâtiment dans Starcraft 2 est un objet qui a un niveau de vie, un nom, il peut produire des unités (via une méthode), etc.

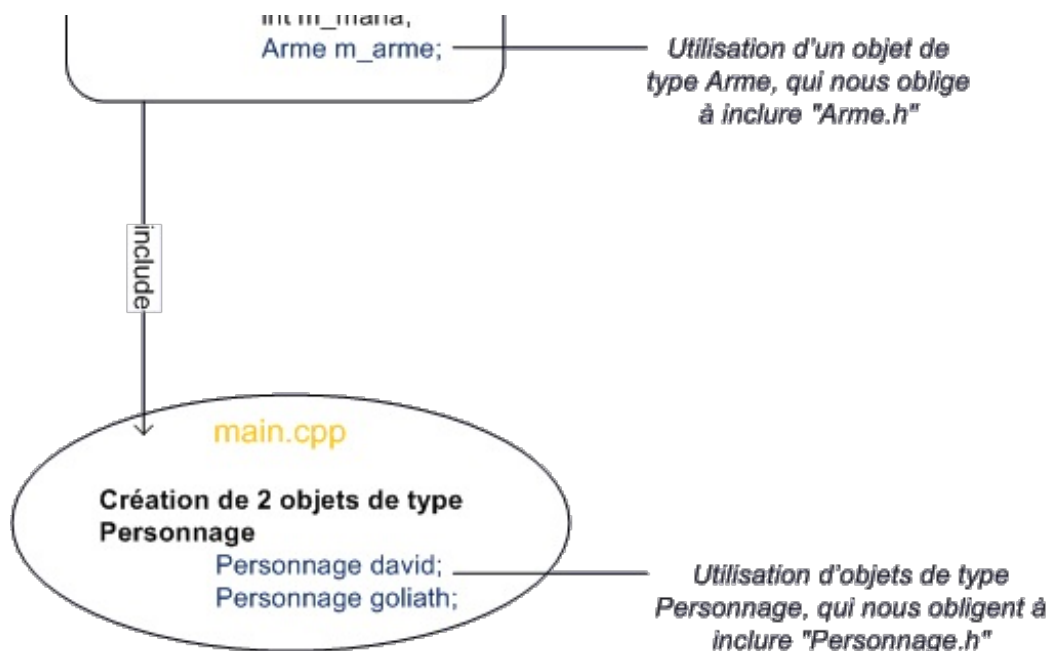
Si vous commencez à voir des objets partout, c'est bon signe ! C'est ce que l'on appelle "penser objet". 😊

Méga schéma résumé

Croyez-moi si vous le voulez, mais je vous demande même pas vraiment d'être capable de programmer tout ce qu'on vient de voir en C++. Je veux que vous reteniez le principe, le concept, comment tout cela est agencé.

Et pour retenir, rien de tel qu'un méga schéma bien mastoc, non ? Ouvrez grand vos yeux, je veux que vous soyez capable de le reproduire les yeux fermés la tête en bas avec du poil à gratter dans le dos !





Si vous avez dû retenir une bonne chose de ce second chapitre, c'est cet échange, cette communication constante entre les objets. Et encore ! On n'avait ici que 2 classes, *Personnage* et *Arme*. Je vous laisse imaginer dans un vrai projet ce que ça donne.



L'intérêt de la POO est là : une organisation précise, chaque objet fait ce qu'il a à faire et délègue certaines parties de son travail à d'autres objets (ici, *Personnage* délèguait la gestion de l'arme à un objet de type *Arme*).

On ne peut pas dire "Je fais de la POO" du jour au lendemain, c'est clair. C'est un travail qui demande de l'organisation, de la méthode. Il faut toujours bien réfléchir avant de se lancer dans un projet, si simple soit-il.

Mais réfléchir un peu avant de programmer, est-ce un mal ? Je ne crois pas. 🤔

Concentrez-vous sur le fichier zip que je vous ai donné et essayez de vous familiariser avec, en faisant par exemple les améliorations proposées. Il ne faut surtout pas que vous soyez perdus.

La surcharge d'opérateurs

On l'a vu, le langage C++ propose beaucoup de fonctionnalités qui peuvent se révéler très utiles, si on arrive à s'en servir correctement.

Une des fonctionnalités les plus étonnantes est la **surcharge des opérateurs**, que nous allons étudier dans ce chapitre. C'est une technique qui permet de réaliser des opérations mathématiques intelligentes entre vos objets lorsque vous utilisez dans votre code des symboles comme +, -, *, etc.

Au final, votre code sera plus court et plus clair, et gagnera donc en lisibilité vous allez voir. 😊

Petits préparatifs

Qu'est-ce que c'est ?

Le principe est très simple. Supposons que vous ayez créé une classe pour stocker une durée (ex : 4h23m), et que vous avez 2 objets de type *Duree*. Vous voulez les additionner entre eux pour connaître la durée totale.

En temps normal, il faudrait créer une fonction "additionner" :

Code : C++

```
resultat = additionner(duree1, duree2);
```

La fonction *additionner* ferait ici la somme de *duree1* et *duree2* et stockerait ça dans *resultat*.

Ça fonctionne, mais ce n'est pas franchement lisible. Ce que je vous propose dans ce chapitre, c'est d'être capable d'écrire ça :

Code : C++

```
resultat = duree1 + duree2;
```

En clair, on fait ici comme si nos objets étaient de simples "nombres". Mais comme un objet c'est plus complexe qu'un nombre (vous avez eu l'occasion de vous en rendre compte 🤔), il va falloir expliquer à l'ordinateur comment effectuer l'opération.

La classe *Duree* pour nos exemples

Toutes les classes ne sont pas forcément adaptées à la surcharge d'opérateurs. Ainsi, ajouter des objets de type *Personnage* entre eux serait pour le moins un peu louche. 🤔

Nous allons donc changer d'exemple, ça sera l'occasion de vous aérer un peu l'esprit sinon vous allez finir par croire que le C++ ne sert qu'à créer des RPG. 😊

Cette classe *Duree* sera capable de stocker des heures, des minutes et des secondes. Rassurez-vous, c'est une classe relativement facile à écrire (plus facile que *Personnage* en tout cas !), ça ne devrait vous poser aucun problème si vous avez compris les chapitres précédents.

Duree.h

Code : C++

```
#ifndef DEF_DUREE
#define DEF_DUREE

class Duree
{
```

```
public:
    Duree(int heures = 0, int minutes = 0, int secondes = 0);

private:
    int m_heures;
    int m_minutes;
    int m_secondes;
};

#endif
```

Chaque objet de type Duree stockera un certain nombre d'heures, minutes et secondes.

Vous noterez que j'ai utilisé des valeurs par défaut au cas où l'utilisateur aurait la flemme de les préciser. 🤖

On pourra donc créer un objet de plusieurs façons différentes :

Code : C++

```
Duree chrono; // Pour stocker 0 heures, 0 minutes et 0 secondes
Duree chrono(5); // Pour stocker 5 heures, 0 minutes et 0 secondes
Duree chrono(5, 30); // Pour stocker 5 heures, 30 minutes et 0
secondes
Duree chrono(0, 12, 55); // Pour stocker 0 heures, 12 minutes et 55
secondes
```

Duree.cpp

L'implémentation de notre constructeur est expédiée en 30 secondes montre en main. 😊

Code : C++

```
#include "Duree.h"

Duree::Duree(int heures, int minutes, int secondes) :
    m_heures(heures), m_minutes(minutes), m_secondes(secondes)
{
}
```


Et dans main.cpp ?

Pour l'instant notre main.cpp ne va déclarer que 2 objets de type Duree, que j'initialise un peu au pif :

Code : C++

```
int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2);

    return 0;
}
```

Voilà, nous sommes prêts à affronter les surcharges d'opérateurs maintenant ! 




Les plus perspicaces d'entre vous auront remarqué que rien ne m'interdit de créer un objet avec 512 minutes et 1455 secondes. En effet, on peut écrire `Duree chrono(0, 512, 1455)`; sans être inquiet. Normalement, cela devrait être interdit, ou tout du moins notre constructeur devrait être assez intelligent pour "découper" les minutes et les convertir en heures/minutes, et de même pour les secondes, afin qu'elles ne dépassent pas 60.

Je ne le fais pas ici, mais je vous encourage à modifier votre constructeur pour faire cette conversion si nécessaire, ça vous entraînera ! Etant donné qu'il faut faire des if et quelques petites opérations mathématiques dans le constructeur, vous ne pourrez pas utiliser que la liste d'initialisation.

Les opérateurs arithmétiques (+, -, *, /, %)

Nous allons commencer par voir les opérateurs mathématiques les plus classiques, à savoir l'addition, la soustraction, la multiplication, la division et le modulo.

Une fois que vous aurez appris à vous servir de l'un d'entre eux, vous verrez que vous saurez vous servir de tous les autres. 

Pour être capable d'utiliser le symbole "+" entre 2 objets, vous devez créer une fonction ayant précisément pour nom `operator+` qui a pour prototype :

Code : C++

```
Objet operator+(Objet const& a, Objet const& b);
```



Même si l'on parle de classe, ceci n'est pas une méthode. C'est une fonction normale située à l'extérieur de toute classe.

La fonction reçoit deux références sur les objets (constantes, donc on ne peut pas les modifier) à additionner. A côté de notre classe `Duree`, on doit donc rajouter cette fonction (ici dans le .h) :

Code : C++

```
Duree operator+(Duree const& a, Duree const& b);
```

C'est la première fois que vous utilisez des références constantes. Dans la [sous-partie sur les références](#), je vous avais expliqué que lors d'un passage par référence, la variable (ou l'objet) n'est pas copié. Notre classe `Duree` contient trois entiers, utiliser une référence permet donc d'éviter la copie inutile de ces trois entiers. Ici, le gain est assez négligeable, mais si vous prenez un objet de type `string`, il peut contenir un très long texte. La copie prendra alors un temps important. C'est pour cela que lorsque l'on manipule des objets, on préfère utiliser des références. Cependant, on aimerait bien que les fonctions ou méthodes ne modifient pas l'objet reçu. C'est pour cela que l'on utilise une référence constante.

Quand on fait `a + b`, `a` et `b` ne doivent pas être modifiés. Le mot-clé `const` est donc essentiel ici.

Mode d'utilisation



Comment ça marche ce truc ?



Dès le moment où vous avez créé cette fonction `operator+`, vous pouvez additionner 2 objets de type `Duree` entre eux :

Code : C++

```
resultat = duree1 + duree2;
```

Ce n'est pas de la magie. En fait le compilateur "traduit" ça par :

Code : C++

```
resultat = operator+(duree1, duree2);
```

... ce qui est beaucoup plus classique et compréhensible pour lui 😊

Il appelle donc la fonction `operator+` en passant `duree1` et `duree2` en paramètre. La fonction, elle, va retourner un résultat de type `Duree`.

Les opérateurs raccourcis

Ce n'est pas le seul moyen d'effectuer une addition ! Rappelez-vous des versions raccourcies des opérateurs. A côté de `+`, il y avait `+=` et de même pour les autres. Contrairement à `+` qui est une fonction, `+=` est une méthode de la classe. Voici son prototype :

Code : C++

```
Duree& operator+=(Duree const& duree);
```

Elle reçoit en argument une autre `Duree` à additionner et renvoie une référence sur l'objet lui-même. Nous verrons plus loin à quoi sert cette référence.

Nous voici donc avec deux manières d'effectuer une addition.

Code : C++

```
resultat = duree1 + duree2; //Utilisation de operator+  
  
duree1 += duree2;           //Utilisation de la méthode operator+=  
de l'objet duree1
```

Vous vous en doutez peut-être les corps de ces fonctions seront très semblables. Si l'on sait faire le calcul avec `+`, il ne faut qu'une petite modification pour obtenir celui de `+=` et vice-versa. C'est somme toute deux fois la même opération mathématique.

Les programmeurs sont des fainéants et écrire deux fois la même fonction est vite ennuyeux. C'est pourquoi on va généralement utiliser une de ces deux opérations pour définir l'autre. Et la règle veut que l'on définisse `operator+` en appelant la méthode `operator+=`.



Mais comment est-ce possible ?

Prenons un exemple plus simple que des `Duree`. Des `int` par exemple et analysons ce qui se passe quand on cherche à les additionner.

Code : C++

```
int a(4), b(5), c(0);  
c = a + b; //c vaut 9
```

On prend la variable `a`, on lui ajoute `b` et on met le tout dans `c`. Ce qui revient presque à écrire :

Code : C++

```
int a(4), b(5), c(0);
a += b;
c = a;    //c vaut 9 mais a vaut maintenant aussi 9
```

La différence étant que dans ce deuxième exemple, la variable `a` a changé de valeur. Si par contre on effectue une copie de `a` avant de la modifier, ce problème disparaît. 🧙

Code : C++

```
int a(4), b(5), c(0);
int copie(a);
copie += b;
c = copie; //c vaut 9 et a vaut toujours 4
```



Le même principe est valable pour `*` et `*=`, `-` et `-=`, etc.

On peut donc effectuer l'opération `+` en faisant une copie suivi d'un `+=`. C'est ce principe que l'on va utiliser pour définir la fonction **operator+** pour notre classe `Duree`. Des fois il faut réfléchir beaucoup pour être fainéant. 🤪

Code : C++

```
Duree operator+(Duree const& a, Duree const& b)
{
    Duree copie(a);
    copie += b;
    return copie;
}
```

Et voilà ! Il ne nous reste plus qu'à définir la méthode **operator+=**. 🤪



Ce passage est peut-être un peu difficile à saisir au premier abord. L'élément important dont il faut se rappeler c'est la manière dont on écrit la définition de **operator+** en utilisant la méthode **operator+=**. Vous pourrez toujours revenir plus tard sur la justification.

Implémentation de `+=`

L'implémentation n'est pas vraiment compliquée, mais il va quand même falloir réfléchir un peu. En effet, ajouter des secondes, minutes et heures ça va, mais il faut faire attention à la retenue si ça dépasse 60.

Je vous recommande d'essayer d'écrire la méthode vous-même, c'est un excellent exercice algorithmique, ça entretient le cerveau, ça vous rend meilleur programmeur (je vous ai convaincus là ? 😊).

Voici ce que donne mon implémentation pour ceux qui ont besoin de la solution :

Code : C++ - Implémentation de l'opérateur `+=`


```

Duree& Duree::operator+=(const Duree &duree2)
{
    // 1 : ajout des secondes
    m_secondes += duree2.m_secondes; // Exceptionnellement autorisé
    car même classe
    // Si le nombre de secondes dépasse 60, on rajoute des minutes
    et on met un nombre de secondes inférieur à 60
    m_minutes += m_secondes / 60;
    m_secondes %= 60;

    // 2 : ajout des minutes
    m_minutes += duree2.m_minutes;
    // Si le nombre de minutes dépasse 60, on rajoute des heures et
    on met un nombre de minutes inférieur à 60
    m_heures += m_minutes / 60;
    m_minutes %= 60;

    // 3 : ajout des heures
    m_heures += duree2.m_heures;

    return *this;
}

```

Ce n'est pas un algorithme ultra-complexe, mais comme je vous avais dit il faut réfléchir un tout petit peu pour pouvoir l'écrire quand même. 😊

Comme nous sommes dans une méthode de la classe, nous pouvons directement modifier les attributs. On va y ajouter les heures, minutes et secondes de l'objet reçu en paramètre, à savoir `duree2`. On a ici exceptionnellement le droit d'accéder directement aux attributs de cet objet car on se trouve dans une méthode de la même classe. C'est un peu tordu mais ça nous aide bien (sinon il aurait fallu créer des méthodes "accesseur" comme `getHeures()`).

Rajouter les secondes, c'est facile. Mais ensuite on doit rajouter un reste si on a dépassé 60 secondes (donc rajouter des minutes). Je ne vous explique pas comment ça fonctionne dans le détail, je vous laisse vous remuer les méninges un peu, ce n'est vraiment pas bien difficile (c'est du niveau des tous premiers chapitres du cours 😊). Vous noterez que c'est un cas où l'opérateur modulo (%), à savoir le reste de la division, est très utile.

Bref, on fait de même avec les minutes, et quant aux heures c'est encore plus facile vu qu'il n'y a pas de reste (on peut dépasser les 24 heures donc pas de problème).

Finalement, il n'y a que la dernière ligne qui devrait vous surprendre. La méthode renvoie l'objet lui-même à l'aide de `*this`. `this` est un mot-clé particulier du langage dont nous reparlerons dans un prochain chapitre. C'est un pointeur vers l'objet qu'on est en train de manipuler. Cette ligne peut-être traduite en français par : "Renvoie l'objet pointé par le pointeur `this`". Les raisons profonde de l'existence de cette ligne ainsi que de la référence comme type de retour sont assez compliquées. Au niveau de ce cours, prenez ça comme une recette de cuisine pour vos opérateurs.

Quelques tests

Pour mes tests, j'ai dû rajouter une méthode `afficher()` à la classe `Duree` (elle fait un cout de la durée tout bêtement).

Voilà mon bôôô main 😊 :

Code : C++

```

#include <iostream>
#include "Duree.h"

using namespace std;

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2);
    Duree resultat;
}

```

```
    duree1.afficher();  
    cout << "+" << endl;  
    duree2.afficher();  
  
    resultat = duree1 + duree2;  
  
    cout << "=" << endl;  
    resultat.afficher();  
  
    return 0;  
}
```

Et le tant attendu résultat à l'écran :

Code : Console

```
0h10m28s  
  
+  
  
0h15m2s  
  
=  
  
0h25m30s
```

Cool, ça marche. 😊

Bon mais ça c'était trop facile, il n'y avait pas de reste dans mon calcul. Corsons un peu les choses avec d'autres valeurs :

Code : Console

```
1h45m50s  
  
+  
  
1h15m50s  
  
=  
  
3h1m40s
```

Yeahhh ! Ça marche ! (et du premier coup pour moi nananère 🤖).

J'ai bien entendu testé d'autres valeurs pour être bien sûr que ça fonctionnait, mais de toute évidence ça marche très bien et mon algo est donc bon. 😊

Bon, on en viendrait presque à oublier l'essentiel dans tout ça. Tout ce qu'on a fait là, c'était pour pouvoir écrire cette ligne :

Code : C++

```
resultat = duree1 + duree2;
```

La surcharge de l'opérateur + nous a permis de rendre notre code clair, simple et lisible, alors qu'on aurait dû utiliser une méthode en temps normal.

N'oublions pas non plus l'opérateur +=. On peut tout à fait l'utiliser directement.

Code : C++

```
#include <iostream>
#include "Duree.h"

using namespace std;

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2);

    duree1.afficher();
    cout << "+" << endl;
    duree2.afficher();

    duree1 += duree2;    //Utilisation directe de l'opérateur +=

    cout << "=" << endl;
    duree1.afficher();

    return 0;
}
```

Ce code affiche bien sûr la même chose que notre premier test. Je vous laisse essayer d'autres valeurs pour vous convaincre que tout est correct. 😊

Télécharger le projet

Pour ceux d'entre vous qui n'auraient pas bien suivi la procédure, ou qui sont tout simplement fainéants (😴), je vous propose de télécharger le projet contenant :

- main.cpp
- Duree.cpp
- Duree.h
- Ainsi que le fichier .cbp de Code::Blocks (si vous utilisez cet IDE comme moi)

[Télécharger le projet \(2 Ko\)](#)

Bonus track #1

Ce qui est vraiment sympa dans tout ça, c'est que tel que notre système est fait, on peut très bien additionner plusieurs durées en même temps sans aucun problème.

Par exemple, je rajoute juste une troisième durée dans mon main et je l'additionne avec les autres :

Code : C++

```
int main()
{
    Duree duree1(1, 45, 50), duree2(1, 15, 50), duree3 (0, 8, 20);
    Duree resultat;

    duree1.afficher();
    cout << "+" << endl;
```

```
    duree2.afficher();  
    cout << "+" << endl;  
    duree3.afficher();  
  
    resultat = duree1 + duree2 + duree3;  
  
    cout << "=" << endl;  
    resultat.afficher();  
  
    return 0;  
}
```

Code : Console

```
1h45m50s  
+  
1h15m50s  
+  
0h8m20s  
=  
3h10m0s
```

C'est cool non vous trouvez pas ?

En fait, la ligne-clé :

Code : C++

```
resultat = duree1 + duree2 + duree3;
```

... revient à écrire :

Code : C++

```
resultat = operator+(operator+(duree1, duree2), duree3);
```

Le tout s'imbrique dans une logique implacable et vient se placer finalement dans l'objet `resultat`. 🤖



Notez que le C++ ne vous permet pas de changer la priorité des opérateurs.

Bonus track #2

Et pour notre seconde bonus track, sachez qu'on n'est pas obligé d'additionner des `Duree` avec des `Duree`, du temps que ça reste logique et compatible.

Par exemple, on pourrait très bien additionner une `Duree` et un `int`. On considérerait dans ce cas que le nombre `int` est un

nombre de secondes à ajouter.

Cela nous permettra d'écrire par exemple :

Code : C++

```
resultat = duree + 30;
```

Vive la surcharge des fonctions et des méthodes! La fonction **operator+** se définit en utilisant le même "truc" qu'avant :

Code : C++

```
Duree operator+(Duree const& duree, int secondes)
{
    Duree copie(duree);
    copie += secondes;
    return copie;
}
```

et tous les calculs sont reportés dans la méthode **operator+=**, comme précédemment.

Code : C++

```
Duree& operator+=(int secondes);
```

... mais vous croyiez tout de même pas que j'allais vous écrire l'implémentation. Allez hop hop hop au boulot ! 🤪

Les autres opérateurs arithmétiques

Maintenant que vous avez vu assez en détail le cas d'un opérateur (celui d'addition pour ceux qui ont la mémoire courte 🤪), vous allez voir que pour la plupart des autres opérateurs c'est très facile et qu'il n'y a pas de difficulté supplémentaire. Le tout est de s'en servir correctement pour la classe que l'on manipule.

Ces opérateurs sont du même "type" que l'addition. Vous les connaissez déjà :

- La soustraction (-)
- La multiplication (*)
- La division (/)
- Le modulo (%), c'est-à-dire le reste de la division

Pour surcharger ces opérateurs, rien de plus simple : créez une fonction dont le nom commence par **operator** suivi de l'opérateur en question. Cela donne donc :

- **operator-** ()
- **operator*** ()
- **operator/** ()
- **operator%** ()

Avec bien sûr les versions raccourcies correspondantes sous forme de méthodes.

- `operator-=()`
- `operator*=()`
- `operator/=()`
- `operator%=()`

Pour notre classe `Duree`, il peut être intéressant de définir la soustraction (`operator-`).

Je vous laisse le soin de le faire, en vous basant sur l'addition ça ne devrait pas être trop compliqué. 😊

En revanche, les autres opérateurs ne servent a priori à rien : en effet, on ne multiplie pas des durées entre elles, et on les divise encore moins. Comme quoi, tous les opérateurs ne sont pas utiles à toutes les classes : ne définissez donc que ceux qui vous seront vraiment utiles.

Si multiplier une `Duree` par une `Duree` n'a pas de sens, en revanche on peut imaginer que l'on multiplie une `Duree` par un nombre entier. Ainsi, l'opération `2h25m50s * 3` est envisageable. Attention à utiliser le bon prototype, en l'occurrence :

Code : C++

```
Duree operator*(Duree const& duree, int nombre);
```

Les opérateurs de flux (<<)

Parmi les nombreuses choses qui ont dû vous choquer quand vous avez commencé le C++, dans la catégorie "*oulah c'est bizarre ça mais on verra plus tard*", il y a l'injection dans les flux d'entrée-sortie. Derrière ce nom barbare se cachent ces petits symboles `>>` et `<<`.

Quand les utilise-t-on ? Allons allons, vous n'allez pas me faire croire que vous avez la mémoire si courte. 😊

Code : C++

```
cout << "Coucou !";
cin >> variable;
```

Figurez-vous justement que `<<` et `>>` sont des opérateurs. Le code ci-dessus revient donc à écrire :

Code : C++

```
operator<<(cout, "Coucou !");
operator>>(cin, variable);
```

On a donc fait appel aux fonctions `operator<<` et `operator>>` ! 😊

Définir ses propres opérateurs pour cout

Nous allons ici nous intéresser plus particulièrement à l'opérateur `<<` utilisé avec `cout`.

Les opérateurs de flux sont définis par défaut pour les types de variables `int`, `double`, `char`, ainsi que pour les objets comme `string`. C'est ainsi que l'on peut aussi bien écrire :

Code : C++

```
cout << "Coucou !";
```

... que :

Code : C++

```
cout << 15;
```

(et c'est là qu'on dit "merci la surcharge des fonctions !" 🤪)

Bon, le problème c'est que `cout` ne connaît pas votre classe flambant neuve `Duree`, et donc qu'il ne possède pas de fonction surchargée pour les objets de ce type. On ne peut donc pas écrire :

Code : C++

```
Duree chrono(0, 2, 30);
cout << chrono; // Erreur : il n'existe pas de fonction
operator<<(cout, Duree &duree)
```

Qu'à cela ne tienne, nous allons écrire cette fonction !



Quoi ?! Mais on ne peut pas modifier le code de la bibliothèque standard ?

Déjà si vous vous êtes posé la question, bravo, c'est que vous commencez à bien vous repérer. En effet, c'est une fonction utilisant un objet de la classe `ostream` (dont `cout` est une instance) que l'on doit définir, et on n'a pas accès au code correspondant.

Lorsque vous incluez `<iostream>`, un objet `cout` est automatiquement déclaré comme ceci :

Code : C++



```
ostream cout;
```

`ostream` est la classe, `cout` est l'objet.

On ne peut pas modifier la classe `ostream`, mais on peut très bien écrire une fonction qui reçoit un de ces objets en argument. Voyons donc comment écrire cette fameuse fonction.

Implémentation d'`operator<<`

Commencez par écrire la fonction :

Code : C++

```
ostream& operator<<( ostream &flux, Duree const& duree )
{
    flux << duree.m_heures << "h" << duree.m_minuttes << "m" <<
    duree.m_secondes << "s"; // Erreur
    return flux;
}
```

Comme vous pouvez le voir, c'est similaire à `operator+`, sauf qu'ici le type de retour est une référence et pas un objet.

Le premier paramètre (référence sur un objet de type `ostream`) qui vous sera automatiquement passé est en fait l'objet `cout` (que l'on appelle ici `flux` dans la fonction pour éviter les conflits de nom). Le second paramètre est une référence constante vers l'objet de type `Duree` que vous tentez d'afficher en utilisant l'opérateur `<<`.

La fonction doit récupérer les attributs qui l'intéressent dans l'objet et les envoyer à l'objet `flux` (qui n'est autre que `cout`). Ensuite, elle retourne une référence sur cet objet, ce qui permet de pouvoir faire une chaîne :

Code : C++

```
cout << duree1 << duree2;
```



Si je compile ça plante ! Ça me dit que je n'ai pas le droit d'accéder aux attributs de l'objet `duree` depuis la fonction !

Eh oui c'est parfaitement normal, car on est à l'**extérieur** de la classe, et les attributs `m_heures`, `m_minutes` et `m_secondes` sont privés. On ne peut donc pas les lire de cet endroit du code.

3 solutions :

- Ou bien vous créez des accesseurs comme on l'a vu (ces fameuses méthodes `getHeures()`, `getMinutes()`, ...), ça marche bien mais c'est un peu ennuyeux à écrire.
- Ou bien utiliser le concept d'amitié, que nous verrons dans un autre chapitre.
- Ou bien vous utilisez la technique que je vais vous montrer. 🤔

On va opter ici pour la troisième solution. 🤖

Changez la 1ère ligne de la fonction comme ceci :

Code : C++

```
ostream &operator<<( ostream &flux, Duree const& duree)
{
    duree.afficher(flux) ; // <- Changement ici
    return flux;
}
```

Et rajoutez une méthode `afficher` dans la classe `Duree`.

Prototype à mettre dans `Duree.h` :

Code : C++

```
void afficher(std::ostream &flux) const;
```

Implémentation de la méthode dans `Duree.cpp` :

Code : C++

```
void Duree::afficher(ostream &flux) const
{
    flux << m_heures << "h" << m_minutes << "m" << m_secondes <<
    "s";
}
```


On passe donc le relai à une méthode à l'intérieur de la classe, qui, elle, a le droit d'accéder aux attributs. La méthode prend en paramètre la référence vers l'objet `flux` pour pouvoir lui envoyer les valeurs qui nous intéressent. Ce qu'on n'a pas pu faire dans la fonction `operator<<`, on le donne à faire à une méthode de la classe `Duree`. Exactement comme pour `operator+` en somme ! On a délégué le travail à une méthode de la classe qui elle a accès aux attributs.

Ouf ! Maintenant dans le main, que du bonheur !

Bon, c'était un peu gymnastique, mais maintenant c'est que du bonheur. 😊

Vous allez pouvoir dans votre main afficher vos objets de type `Duree` très simplement :

Code : C++

```
int main()
{
    Duree duree1(2, 25, 28), duree2(0, 16, 33);

    cout << duree1 << " et " << duree2 << endl;

    return 0;
}
```

Résultat :

Code : Console

```
2h25m28s et 0h16m33s
```

Enfantin. 😊

Comme quoi, on prend un peu de temps pour écrire la classe, mais ensuite quand on doit l'utiliser c'est extrêmement simple !

Et l'on peut même combiner nos opérateurs dans une seule expression. Faire une addition et afficher le résultat directement :

Code : C++

```
int main()
{
    Duree duree1(2, 25, 28), duree2(0, 16, 33);

    cout << duree1 + duree2 << endl;

    return 0;
}
```

Comme pour les `int`, `double`, etc. Nos objets deviennent réellement simples à utiliser.

Les opérateurs de comparaison (==, >, <, ...)

Ces opérateurs vont vous permettre de comparer des objets entre eux. Le plus utilisé d'entre eux est probablement l'opérateur d'égalité (==) qui permet de vérifier si 2 objets sont égaux. C'est à vous d'écrire le code de la méthode qui détermine si les objets sont identiques, l'ordinateur ne peut pas le deviner pour vous car il ne connaît pas la "logique" de vos objets. 😊

Tous ces opérateurs de comparaison ont un point en commun particulier : ils renvoient un booléen (`bool`). C'est normal, ces opérateurs répondent à des questions du type "a est-il plus grand que b ?" ou "a est-il égal à b ?"

L'opérateur ==

On va écrire l'implémentation de l'opérateur d'égalité pour commencer. Vous allez voir qu'on va beaucoup s'inspirer de la technique utilisée pour l'opérateur <<. Le recyclage des idées c'est bien. 😊

Code : C++

```
bool operator==(Duree const& a, Duree const& b)
{
    //Teste si a.m_heure == b.m_heure etc.
    if (a.m_heures == b.m_heures && a.m_minutes == b.m_minutes &&
        a.m_secondes == b.m_secondes)
        return true;
    else
        return false;
}
```

On compare à chaque fois un attribut de l'objet dans lequel on se trouve avec un attribut de l'objet auquel on se compare (les heures avec les heures, les minutes avec les minutes...). Si ces 3 valeurs sont identiques alors on peut considérer que les objets sont identiques et renvoyer **true** (vrai).

Sauf qu'il y a un petit souci. Il nous faudrait lire les attributs des objets a et b. Comme le veut la règle, ils sont privés et donc inaccessible depuis l'extérieur de la classe. Appliquons donc la même stratégie que pour l'opérateur <<.

On commence par créer une méthode `estEgal()` qui renvoie **true** si b est égal à l'objet dont on a appelé la méthode.

Code : C++

```
bool Duree::estEgal(Duree const& b) const
{
    //Teste si a.m_heure == b.m_heure etc.
    if (m_heures == b.m_heures && m_minutes == b.m_minutes &&
        m_secondes == b.m_secondes)
        return true;
    else
        return false;
}
```

Et on utilise cette méthode dans notre opérateur d'égalité :

Code : C++

```
bool operator==(Duree const& a, Duree const& b)
{
    return a.estEgal(b);
}
```

Dans le main, on peut faire un simple test de comparaison pour vérifier si on a fait les choses correctement :

Code : C++

```
int main()
{
    Duree duree1(0, 10, 28), duree2(0, 10, 28);

    if (duree1 == duree2)
        cout << "Les durees sont identiques";
    else
        cout << "Les durees sont differentes";
}
```

```
    return 0;
}
```

Résultat :

Code : Console

```
Les durees sont identiques
```

L'opérateur !=

Tester l'égalité, c'est bien, mais parfois on aime savoir si deux objets sont différents. On écrit alors un opérateur !=. Celui-là, il est très simple à écrire. 😊 Pour tester si deux objets sont différents, il suffit de tester si ils ne sont pas égaux !

Code : C++

```
bool operator!=(Duree const& a, Duree const& b)
{
    if(a == b)           //On utilise l'opérateur == qu'on a défini
        précédemment !
        return false;    //Si ils sont égaux, alors ils ne sont pas
        différents
    else
        return true;     //Et si ils ne sont pas égaux, c'est qu'ils
        sont différents ;-)
}
```

Je vous avais dit que ce serait facile. Réutiliser des opérateurs déjà écrits est une bonne habitude à prendre. D'ailleurs on l'avait déjà fait pour + qui utilisait +=.

L'opérateur <

Je vous préviens on va pas tous les faire sinon on y est encore demain. 😊

Si l'opérateur == peut s'appliquer à la plupart des objets, il n'est pas certain que l'on puisse dire de tous nos objets qui est le plus grand. Tous n'ont pas forcément une notion de grandeur, prenez par exemple notre classe `Personnage`, il serait je pense assez stupide de vouloir vérifier si un `Personnage` est "inférieur" à un autre ou non (à moins que vous ne compariez les vies... à vous de voir).

En tout cas avec la classe `Duree` on a de la chance, il est facile et "logique" de vérifier si une `Duree` est inférieure à une autre.

Voici mon implémentation pour l'opérateur "est strictement inférieur à" (<) :

Code : C++

```
bool operator<(Duree const &a, Duree const& b)
{
    if(a.estPlusPetitQue(b))
        return true;
    else
        return false;
}
```

Et la méthode `estPlusPetitQue()` de la classe `Duree` :

Code : C++

```
bool Duree::estPlusPetitQue(Duree const& b) const
{
    if (m_heures < b.m_heures)
        return true;
    else if (m_heures == b.m_heures && m_minutes < b.m_minutes)
        return true;
    else if (m_heures == b.m_heures && m_minutes == b.m_minutes &&
m_secondes < b.m_secondes)
        return true;
    else
        return false;
}
```

Avec un peu de réflexion on finit par trouver cet algorithme, il suffit d'activer un peu ses méninges. 😊

Vous noterez que la méthode renvoie `false` si les durées sont identiques : c'est normal, car il s'agit de l'opérateur "strictement inférieur à" (<). En revanche, si ça avait été la méthode de l'opérateur "inférieur ou égal à" (<=), il aurait fallu renvoyer `true`.

Je vous laisse le soin de tester dans le main si ça fonctionne correctement. 😊

Les autres opérateurs de comparaison

On ne va pas les écrire ici, ça surchargerait inutilement. Mais comme pour `!=` et `==`, il suffit d'utiliser correctement `<` pour tous les implémenter. Je vous invite à essayer de les implémenter pour notre classe `Duree`, ça fera un bon exercice sur le sujet. Il reste notamment :

- `operator>()`
- `operator<=()`
- `operator>=()`

Si vous avez un peu du mal à vous repérer dans le code, ce que je peux comprendre, je mets à votre disposition le projet complet comme tout à l'heure dans ce zip :

Télécharger les sources (2 Ko)

Il y a énormément d'autres opérateurs surchargeables en C++, en fait presque tout peut être surchargé. Chaque opérateur étant particulier, il serait impossible de tout voir dans ce chapitre. Au moins avons-nous pu voir les principaux. 😊

A titre d'information, sachez qu'il est aussi possible de surcharger :

- `new` et `delete` : l'allocation dynamique, s'il y a besoin de faire des vérifications spéciales lors d'une allocation de mémoire
- `&` et `*` : opérateurs d'indirection et de déréférencement pour manipuler les pointeurs
- `++` et `--` : opérateurs d'incrément et de décrément
- `[]` : pour parcourir l'objet comme un tableau. Le type `string` s'en sert d'ailleurs pour que l'on puisse écrire `monString[3]` et ainsi accéder au 4ème caractère comme si c'était un tableau, alors que c'est en fait un objet. Malin, il fallait y penser !
- etc.

Bref, vous l'aurez compris, la surcharge des opérateurs est un outil puissant, pour ne pas dire très puissant si on commence à s'en servir sur l'allocation dynamique ou encore les opérateurs d'indirection et de déréférencement.

Mon conseil serait : ne faites la surcharge que si elle vous sera vraiment utile. C'est certes un outil puissant, mais il n'est pas nécessaire de le mettre à toutes les sauces. Votre classe doit proposer des fonctionnalités utiles et non pas farfelues !

TP: La POO en pratique avec ZFraction

Vous avez appris dans les chapitres précédents à manipuler des classes et même à en créer par vous-même. Il est donc grand temps de mettre tout ça en pratique avec un TP. Vous allez devoir **écrire une classe complète**, vous allez voir, c'est le meilleur moyen de digérer toutes ces nouvelles notions.

C'est le premier TP sur la POO, il porte donc sur les bases. C'est donc le bon moment d'arrêter un peu la lecture du cours, de souffler un peu et d'essayer de réaliser cet exercice par vous-même. Vous aurez aussi l'occasion de vérifier vos connaissances et donc de retourner lire les chapitres sur les éléments qui vous ont manqués.

Le sujet de ce TP devrait vous rappeler vos cours de mathématiques du collège. ~~Nous allons~~ Vous allez écrire une classe représentant la notion de **fraction**. Le C++ permet d'utiliser des nombres entiers via le type `int`, des nombres réels via le type `double`, mais il n'offre rien pour les nombre rationnels. À vous de pallier ce manque ! 😊

$$\frac{1}{2} + \frac{5}{3} = \frac{13}{6}$$

$$\frac{1}{2} \times \frac{5}{3} = \frac{5}{6}$$

Préparatifs et conseils

La classe que nous allons réaliser n'est pas très compliquée. Et il est assez aisé d'imaginer quelles méthodes et opérateurs nous allons utiliser. Cet exercice va en particulier tester vos connaissances sur :

- Les attributs et leurs droits d'accès.
- Les constructeurs.
- La surcharge des opérateurs.

C'est donc le dernier moment de réviser ! 🧐

Description de la classe ZFraction

Commençons par choisir un nom pour notre classe. Ce serait bien qu'il contienne le mot "fraction" et comme vous êtes sur le site du zéro, je vous propose d'ajouter un "Z" au début. Ce sera donc ZFraction. Ce n'est pas super original, mais au moins on sait directement à quoi on a affaire. 😊

Utiliser des `int` ou des `double` est très simple. On les déclare, on les initialise et on utilise ensuite les opérateurs comme sur une calculatrice. Ce serait vraiment super de pouvoir faire la même chose pour des fractions.

On aimerait donc bien que le `main()` suivant compile et fonctionne correctement :

Code : C++ - Utilisation de ZFraction

```
#include <iostream>
#include "ZFraction.h"
using namespace std;

int main()
{
    ZFraction a(4,5);           //Déclare une fraction valant 4/5
    ZFraction b(2);             //Déclare une fraction valant 2/1 (ce
    //qui vaut 2)
    ZFraction c,d;              //Déclare deux fractions valant 0

    c = a+b;                    //Calcule 4/5 + 2/1 = 14/5

    d = a*b;                    //Calcule 4/5 * 2/1 = 8/5

    cout << a << " + " << b << " = " << c << endl;

    cout << a << " * " << b << " = " << d << endl;
```

```

    if(a > b)
        cout << "a est plus grand que b." << endl;
    else if(a==b)
        cout << "a est egal a b." << endl;
    else
        cout << "a est plus petit que b." << endl;

    return 0;
}

```

Et voici le résultat espéré :

Code : Console

```

4/5 + 2 = 14/5
4/5 * 2 = 8/5
a est plus petit que b.

```

Pour arriver à cela, il nous faudra donc :

- Écrire la classe avec ses attributs.
- Réfléchir aux constructeurs à implémenter.
- Surcharger les opérateurs +, *, <<, < et == (au moins).




En maths, lorsque l'on manipule des fractions, on utilise toujours des fractions simplifiées. C'est-à-dire que l'on écrira $\frac{4}{5}$ plutôt que $\frac{8}{10}$ même si ces deux fractions ont la même valeur. Il faudra donc faire en sorte que notre classe `ZFraction` respecte cette règle.

Si vous vous sentez prêt, alors allez-y ! Je n'ai rien de plus à ajouter concernant la donnée. Vous pourrez trouver certains rappels sur les calculs avec les nombres rationnels dans le [cours de maths](#) disponible sur ce site.

Si par contre vous avez peur de vous lancer seul, je vous propose de vous accompagner pour les premiers pas.

Créer un nouveau projet

Pour faire ce TP, vous allez devoir créer un nouveau projet. Utilisez l'IDE que vous voulez, moi pour ma part vous savez que j'utilise Code::Blocks 

Demandez à créer un nouveau **projet console C++**.

Ce projet sera constitué de 3 fichiers que vous pouvez déjà créer :

- **main.cpp** : ce fichier contiendra uniquement la fonction main. Dans la fonction main, nous créerons des objets basés sur notre classe `ZFraction` pour tester son fonctionnement. A la fin, votre fonction `main()` devra ressembler à celle que je vous ai montré plus haut.
- **ZFraction.h** : ce fichier contiendra le prototype de notre classe `ZFraction` avec la liste de ses attributs et les prototypes de ses méthodes.
- **ZFraction.cpp** : ce fichier contiendra l'implémentation des méthodes de la classe `ZFraction`, c'est-à-dire le "code" à l'intérieur des méthodes.



Faites attention aux noms des fichiers et en particulier aux majuscules et minuscules. Les fichiers `ZFraction.h` et `ZFraction.cpp` commencent par 2 lettres majuscules, si vous écrivez "zfraction" ou encore "Zfraction" ça ne marchera pas et vous aurez des problèmes.

Le code de base des fichiers

Nous allons écrire un peu de code dans chacun de ces fichiers. Juste le strict minimum pour pouvoir commencer.

main.cpp

Bon, celui-là, je vous l'ai déjà donné. 😊

Mais pour commencer en douceur, je vous propose de simplifier l'intérieur de la fonction `main()` et d'y ajouter des instructions petit-à-petit au fur et à mesure de l'avancement de votre classe.

Code : C++ - main.cpp

```
#include <iostream>
#include "ZFraction.h"
using namespace std;

int main()
{
    ZFraction a(1,5); // Crée une fraction valant 1/5
    return 0;
}
```

Pour l'instant, on se contente d'un appel au constructeur de `ZFraction`. Pour le reste, on verra plus tard.

ZFraction.h

Ce fichier contiendra la définition de la classe `ZFraction`. Il inclut aussi `iostream` pour nos besoins futurs (nous aurons besoin de faire des `cout` dans la classe les premiers temps, ne serait-ce que pour déboguer notre classe).

Code : C++ - ZFraction.h

```
#ifndef DEF_FRACTION
#define DEF_FRACTION

#include <iostream>

class ZFraction
{
public:

private:

};

#endif
```

Pour l'instant, la classe est encore vide. Je ne vais pas non plus tout vous faire hein ! J'y ai quand même mis une partie privée et une partie publique. Souvenez-vous de la règle principale de la POO qui veut que tous les attributs soient dans la partie privée. Je vous en voudrais beaucoup si vous ne la respectiez pas. 😞



Comme tous les fichiers `.h`, `ZFraction.h` contient deux lignes commençant par `#` au début du fichier et une autre tout à la fin. Code::Blocks crée automatiquement ces lignes. Si votre IDE ne le fait pas, pensez à les ajouter. Elles évitent bien des soucis de compilation.

ZFraction.cpp

C'est le fichier qui va contenir les définitions des méthodes. Comme notre classe est encore vide, il n'y a donc rien à y écrire. Il faut juste penser à inclure l'entête `ZFraction.h`.

Code : C++ - `ZFraction.cpp`

```
#include "ZFraction.h"
```

Nous voilà enfin prêt à attaquer la programmation !

Choix des attributs de la classe

La première étape de la création d'une classe est souvent le choix des attributs. Il faut se demander de quelles briques de base notre classe est constituée. Avez-vous une petite idée ?

Voyons ça ensemble. Un nombre rationnel est composé de deux nombres entiers appelés le numérateur (celui qui est au-dessus de la barre de fraction) et le dénominateur (celui du dessous). Cela nous fait donc deux constituants. Les nombres entiers en C++ s'appellent des `int`. Ajoutons donc deux `int` à notre classe :

Code : C++ - `ZFraction.h`

```
#ifndef DEF_FRACTION
#define DEF_FRACTION

#include <iostream>

class ZFraction
{
public:

private:

    int m_numérateur;        //Le numérateur de la fraction
    int m_dénominateur;      //Le dénominateur de la fraction

};

#endif
```

Nos attributs commencent toujours par le préfixe `"m_"`. C'est une bonne habitude de programmation que je vous ai enseignée dans les chapitres précédents 😊

Cela nous permettra par la suite de savoir si on est en train de manipuler un attribut de la classe ou une simple variable "locale" à une méthode.

Les constructeurs

Je ne vais pas tout vous dire non plus, mais dans le `main()` d'exemple que je vous ai présenté tout au début, on utilisait trois constructeurs différents :

- Le premier recevait deux entiers comme argument. Ils représentaient respectivement le numérateur et le dénominateur de la fraction. C'est sans doute le plus intuitif des trois à écrire.
- Le deuxième constructeur prend un seul entier en argument et construit une fraction égale à ce nombre entier. Cela veut dire que le dénominateur vaut 1 dans ce cas.
- Finalement, le dernier constructeur ne prend aucun argument (constructeur par défaut) et crée une fraction valant 0.

Je ne vais rien expliquer de plus. Je vous propose de commencer par écrire au moins le premier de ces trois constructeurs. Les autres suivront rapidement, j'en suis sûr.

Les opérateurs

La part la plus importante de ce TP sera l'implémentation des opérateurs. Il faut bien réfléchir à la manière de les écrire. Vous pouvez bien sûr vous inspirer de ce qui a été fait pour la classe `Duree` du chapitre précédent. Par exemple, utiliser la méthode `operator+=` pour définir l'opérateur `+`. Ou écrire une méthode `estEgaleA()` pour l'opérateur d'égalité.

Une bonne chose à faire est de commencer par l'opérateur `<<`. Vous pourrez alors facilement tester vos autres opérateurs.

Simplifier les fractions

L'important avec les fractions est de toujours manipuler des fractions simplifiées. C'est-à-dire que l'on va préférer écrire $\frac{2}{5}$ que $\frac{4}{10}$ par exemple. Il serait bien que notre classe fasse de même et simplifie elle-même la fraction qu'elle représente.

Il nous faut donc un moyen mathématique de le faire puis traduire le tout en C++. Si l'on a une fraction $\frac{a}{b}$, il faut calculer le plus grand commun diviseur de a et b puis diviser a et b par ce nombre. Par exemple, le pgcd de 4 et 10 est 2, ce qui veut dire que l'on peut simplifier les numérateurs et dénominateurs de $\frac{4}{10}$ par 2, ce qui nous fait bien $\frac{2}{5}$.

Calculer le pgcd n'est pas une opération facile. Je vous propose donc une fonction pour le faire. Je vous invite à l'ajouter dans votre fichier `ZFraction.cpp`.

Code : C++ - Fonction de calcul du pgcd

```
int pgcd(int a, int b)
{
    while (b != 0)
    {
        const int t = b;
        b = a%b;
        a=t;
    }
    return a;
}
```

Et à ajouter le prototype correspondant dans `ZFraction.h`:

Code : C++ - ZFraction.h

```
#ifndef DEF_FRACTION
#define DEF_FRACTION

#include <iostream>

class ZFraction
{
    //Contenu de la classe...
};

int pgcd(int a, int b);

#endif
```

Vous pourrez alors utiliser cette fonction dans les méthodes de la classe.

Allez au boulot ! 😊

Correction

TEEEERRRRMINNÉÉÉÉ ! Lâchez vos claviers, le temps imparti est écoulé. 😊

Il est temps de passer à la phase de correction. Vous avez certainement passé pas mal de temps à réfléchir aux différentes méthodes, opérateurs et autres ~~horreurs~~ joyeusetés du C++. Si vous n'avez pas réussi à tout faire, ce n'est pas grave. Lire la correction pour saisir les grands principes devrait vous aider. Et puis vous saurez peut-être vous rattraper avec les améliorations proposées en fin de chapitre.

Je vous propose de passer tranquillement en revue les différentes étapes de création de la classe.

Les constructeurs

Je vous avais suggéré de commencer par le constructeur prenant en argument deux entiers, le numérateur et le dénominateur. Voici ma version.

Code : C++ - Constructeur prenant deux entiers

```
ZFraction::ZFraction(int num, int den)
    :m_numérateur(num), m_dénominateur(den)
{
}
```

On utilise la liste d'initialisation pour remplir les attributs `m_numérateur` et `m_dénominateur` de la classe. Jusque-là, rien de sorcier.

En continuant sur cette lancée, on peut écrire les deux autres constructeurs :

Code : C++ - Les autres constructeurs

```
ZFraction::ZFraction(int entier)
    :m_numérateur(entier), m_dénominateur(1)
{
}

ZFraction::ZFraction()
    :m_numérateur(0), m_dénominateur(1)
{
}
```

Il fallait se rappeler que le nombre **5** s'écrit comme la fraction $\frac{5}{1}$ et **0** comme $\frac{0}{1}$.

Le cahier des charges est donc rempli dans ce domaine. Avant de commencer à faire des choses compliquées, écrivons l'opérateur `<<` pour l'affichage de notre fraction. On pourra ainsi facilement voir ce qui se passe dans notre classe en cas d'erreur.

Afficher une fraction

Comme nous l'avons vu dans le chapitre sur les opérateurs, la meilleure solution est d'utiliser une méthode `affiche()` dans la classe et d'appeler cette méthode dans la fonction `operator<<`. Un "copier-coller" du chapitre précédent nous donne donc directement le code de l'opérateur.

Code : C++ - Opérateur d'injection dans un flux

```
ostream& operator<<(ostream& flux, ZFraction const& fraction)
{
    fraction.affiche(flux);
    return flux;
}
```

Et pour la méthode `affiche()`, je vous propose cette version :

Code : C++ - Méthode d'affichage d'une fraction

```
void ZFraction::affiche(ostream& flux) const
{
    if(m_denominateur == 1)
    {
        flux << m_numérateur;
    }
    else
    {
        flux << m_numérateur << '/' << m_denominateur;
    }
}
```



Notez le **const** dans le prototype de la méthode. Il montre que `affiche()` ne modifiera pas l'objet. Normal, puisque nous ne faisons qu'afficher ses propriétés.

Vous avez certainement écrit quelque chose d'approchant. J'ai distingué le cas où le dénominateur vaut 1. Une fraction dont le dénominateur vaut 1 est un nombre entier. On a donc pas besoin d'afficher la barre de fraction et le dénominateur. Mais c'est juste une question d'esthétique. 😊

L'opérateur d'addition

Comme pour `<<`, le mieux est d'employer la recette du chapitre précédent. Définir une méthode `operator+=()` dans la classe et l'utiliser dans la fonction `operator+()`.

Code : C++ - Opérateur d'addition raccourci

```
ZFraction operator+(ZFraction const& a, ZFraction const& b)
{
    ZFraction copie(a);
    copie+=b;
    return copie;
}
```

La difficulté réside dans l'implémentation de l'opérateur d'addition raccourci. Comme toujours. 😊

En ressortant mes cahiers de maths, j'ai retrouvé la formule d'addition de deux fractions :

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d}$$

Ce qui donne en C++ :

Code : C++ - Opérateur d'addition

```

ZFraction& ZFraction::operator+=(ZFraction const& autre)
{
    m_numérateur = autre.m_dénominateur * m_numérateur +
m_dénominateur * autre.m_numérateur;
    m_dénominateur = m_dénominateur * autre.m_dénominateur;

    return *this;
}

```



Comme tous les opérateurs raccourcis, l'opérateur += doit renvoyer une référence sur *this. C'est une convention.

L'opérateur de multiplication

La formule de multiplication de deux fractions est plus simple encore que l'addition :

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$

Je vais garder mes livres maths à portée de main je crois... 😊

Et je ne vais pas vous surprendre si je vous dis qu'il faut utiliser la méthode `operator*=()` et la fonction `operator*()`. Je crois qu'on commence à comprendre le truc. 😊

Code : C++ - Opérateurs de multiplication

```

ZFraction operator*(ZFraction const& a, ZFraction const& b)
{
    ZFraction copie(a);
    copie*=b;
    return copie;
}

ZFraction& ZFraction::operator*=(ZFraction const& autre)
{
    m_numérateur *= autre.m_numérateur;
    m_dénominateur *= autre.m_dénominateur;

    return *this;
}

```

Les opérateurs de comparaison

Comparer des fractions pour tester si elles sont identiques revient à tester si leurs numérateurs et dénominateurs sont égaux. L'algorithme est donc à nouveau relativement simple. Je vous propose, comme toujours, de passer par une méthode de la classe puis d'utiliser cette méthode dans les opérateurs externes.

Code : C++ - Opérateurs de comparaison

```

bool ZFraction::estEgal(ZFraction const& autre) const
{
    if(m_numérateur == autre.m_numérateur && m_dénominateur ==
autre.m_dénominateur)
        return true;
    else
        return false;
}

```

```

    }

    bool operator==(ZFraction const& a, ZFraction const& b)
    {
        if(a.estEgal(b))
            return true;
        else
            return false;
    }

    bool operator!=(ZFraction const& a, ZFraction const& b)
    {
        if(a.estEgal(b))
            return false;
        else
            return true;
    }

```

Une fois que la méthode `estEgal()` est implémentée, on a deux opérateurs pour le prix d'un seul. Parfait, je n'avais pas envie de réfléchir deux fois. 🤖

Les opérateurs d'ordre

Il ne nous reste plus qu'à écrire un opérateur permettant de vérifier si une fraction est plus petite que l'autre. Il y a plusieurs moyens de faire ça. Toujours dans mes livres de maths, j'ai retrouvé une vieille relation intéressante :

$$\frac{a}{b} < \frac{c}{d} \iff a \cdot d < b \cdot c$$

Cette relation peut être traduite en C++ pour obtenir le corps de la méthode `estPlusPetitQue()` :

Code : C++ - Méthode de comparaison de deux fractions

```

bool ZFraction::estPlusPetitQue(ZFraction const& autre) const
{
    if(m_numerator * autre.m_denominateur < m_denominateur *
    autre.m_numerator)
        return true;
    else
        return false;
}

```

Et cette fois, ce n'est pas un pack "2 en 1", mais "4 en 1". Avec un peu de réflexion, on peut utiliser cette méthode pour les opérateurs `<`, `>`, `<=` et `>=`. 🧙

Code : C++ - Les opérateurs de comparaison

```

bool operator<(ZFraction const& a, ZFraction const& b) //Vrai si a<b
    donc si a est plus petit que b
{
    if(a.estPlusPetitQue(b))
        return true;
    else
        return false;
}

bool operator>(ZFraction const& a, ZFraction const& b) //Vrai si a>b
    donc si b est plus petit que a
{

```

```

        if(b.estPlusPetitQue(a))
            return true;
        else
            return false;
    }

    bool operator<=(ZFraction const& a, ZFraction const& b) //Vrai si
    a<=b donc si b n'est pas plus petit que a
    {
        if(b.estPlusPetitQue(a))
            return false;
        else
            return true;
    }

    bool operator>=(ZFraction const& a, ZFraction const& b) //Vrai si
    a>=b donc si a n'est pas plus petit que b
    {
        if(a.estPlusPetitQue(b))
            return false;
        else
            return true;
    }
}

```

Avec ces quatre derniers opérateurs, nous avons fait le tour de ce qui était demandé. Ou presque. Il nous reste à voir la partie la plus difficile : le problème de la simplification des fractions.

Simplifier les fractions

Je vous ai expliqué dans la présentation du problème quel algorithme utiliser pour simplifier une fraction. Il faut calculer le pgcd du numérateur et du dénominateur. Puis diviser les deux attributs de la fraction par ce nombre.

Comme c'est une opération qui doit être exécutée à différents endroits, je vous propose d'en faire une méthode de la classe. On aura ainsi pas besoin de récrire l'algorithme à différents endroits. Cette méthode n'a pas à être appelée par les utilisateurs de la classe. C'est de la mécanique interne. Elle va donc dans la partie privée de la classe.

Code : C++ - La méthode simplifie()

```

void ZFraction::simplifie()
{
    int nombre=pgcd(m_numérateur, m_dénominateur); //Calcul du pgcd

    m_numérateur /= nombre;    //Et on simplifie
    m_dénominateur /= nombre;
}

```



Quand faut-il utiliser cette méthode ?

Bonne question ! Mais vous devriez avoir la réponse. 😊

Il faut simplifier la fraction à chaque fois qu'un calcul est effectué. C'est-à-dire, dans les méthodes **operator+=()** et **operator*=()** :

Code : C++

```

ZFraction& ZFraction::operator+=(ZFraction const& autre)
{
    m_numérateur = autre.m_dénominateur * m_numérateur +

```

```

    m_denominateur * autre.m_numérateur;
    m_denominateur = m_denominateur * autre.m_denominateur;

    simplifie();    //On simplifie la fraction
    return *this;
}

ZFraction& ZFraction::operator*=(ZFraction const& autre)
{
    m_numérateur *= autre.m_numérateur;
    m_denominateur *= autre.m_denominateur;

    simplifie();    //On simplifie la fraction
    return *this;
}

```

Mais ce n'est pas tout ! Quand l'utilisateur construit une fraction, rien ne garantit qu'il le fait correctement. Il peut très bien initialiser sa ZFraction avec les valeurs $\frac{4}{8}$ par exemple. Il faut donc aussi appeler la méthode dans le constructeur qui prend deux arguments.

Code : C++ - Constructeur prenant deux entiers

```

ZFraction::ZFraction(int num, int den)
    :m_numérateur(num), m_denominateur(den)
{
    simplifie(); //On simplifie au cas où l'utilisateur aurait
    entrée de mauvaises informations
}

```

Et voilà ! En fait, si vous regardez bien, nous avons dû ajouter un appel à la méthode `simplifie()` dans toutes les méthodes qui ne sont pas déclarées constantes ! Chaque fois que l'objet est modifié, il faut simplifier la fraction. On aurait pu éviter de réfléchir et simplement analyser notre code à la recherche de ces méthodes. Utiliser `const` est donc un atout de sécurité. On voit tout de suite où il faut faire des vérifications (appeler `simplifie()`) et où c'est inutile.

Notre classe est maintenant fonctionnelle et respecte les critères que je vous ai imposé. Hip Hip Hip Hourra !

Code complet

Pour finir, je vous propose le code complet de la classe.

Code : C++ - ZFraction.h

```

#ifndef DEF_FRACTION
#define DEF_FRACTION

#include <iostream>

class ZFraction
{
public:

    //Constructeurs
    ZFraction(int num, int den);
    ZFraction(int nombre);
    ZFraction();

    //Affichage
    void affiche(std::ostream& flux) const;
}

```



```

//Opérateurs arithmétiques
ZFraction& operator+=(ZFraction const& autre);
ZFraction& operator*=(ZFraction const& autre);

//Méthodes de comparaison
bool estEgal(ZFraction const& autre) const;
bool estPlusPetitQue(ZFraction const& autre) const;

private:

    int m_numerator;        //Le numérateur de la fraction
    int m_denominateur;     //Le dénominateur de la fraction

    // Simplifie une fraction
    void simplifie();

};

//Opérateur d'injection dans un flux
std::ostream& operator<<(std::ostream& flux, ZFraction const&
fraction);

//Opérateurs arithmétiques
ZFraction operator+(ZFraction const& a, ZFraction const& b);
ZFraction operator*(ZFraction const& a, ZFraction const& b);

//Opérateurs de comparaison
bool operator==(ZFraction const& a, ZFraction const& b);
bool operator!=(ZFraction const& a, ZFraction const& b);
bool operator<(ZFraction const& a, ZFraction const& b);
bool operator>(ZFraction const& a, ZFraction const& b);
bool operator>=(ZFraction const& a, ZFraction const& b);
bool operator<=(ZFraction const& a, ZFraction const& b);

//Calcul du pgcd
int pgcd(int a, int b);

#endif

```

Code : C++ - ZFraction.cpp

```

#include "ZFraction.h"
using namespace std;

//Constructeurs
ZFraction::ZFraction(int num, int den)
    :m_numerator(num), m_denominateur(den)
{
    simplifie();
}

ZFraction::ZFraction(int entier)
    :m_numerator(entier), m_denominateur(1)
{
}

ZFraction::ZFraction()
    :m_numerator(0), m_denominateur(1)
{
}

//Affichage
void ZFraction::affiche(ostream& flux) const
{
    if(m_denominateur == 1)
    {
        flux << m_numerator;
    }
}

```

```

        flux << m_numerator;
    }
    else
    {
        flux << m_numerator << '/' << m_denominateur;
    }
}

//Opérateur d'injection dans un flux
ostream& operator<<(ostream& flux, ZFraction const& fraction)
{
    fraction.affiche(flux);
    return flux;
}

//Opérateurs arithmétiques
ZFraction& ZFraction::operator+=(ZFraction const& autre)
{
    m_numerator = autre.m_denominateur * m_numerator +
m_denominateur * autre.m_numerator;
    m_denominateur = m_denominateur * autre.m_denominateur;

    simplifie();
    return *this;
}

ZFraction& ZFraction::operator*=(ZFraction const& autre)
{
    m_numerator *= autre.m_numerator;
    m_denominateur *= autre.m_denominateur;

    simplifie();
    return *this;
}

//Méthodes de comparaison
bool ZFraction::estEgal(ZFraction const& autre) const
{
    if(m_numerator == autre.m_numerator && m_denominateur ==
autre.m_denominateur)
        return true;
    else
        return false;
}

bool ZFraction::estPlusPetitQue(ZFraction const& autre) const
{
    if(m_numerator * autre.m_denominateur < m_denominateur *
autre.m_numerator)
        return true;
    else
        return false;
}

//Simplification
void ZFraction::simplifie()
{
    int nombre=pgcd(m_numerator, m_denominateur); //Calcul du pgcd

    m_numerator /= nombre;      //Et on simplifie
    m_denominateur /= nombre;
}

//Opérateurs externes
ZFraction operator+(ZFraction const& a, ZFraction const& b)
{
    ZFraction copie(a);
    copie+=b;
    return copie;
}

```

```
ZFraction operator*(ZFraction const& a, ZFraction const& b)
{
    ZFraction copie(a);
    copie*=b;
    return copie;
}

bool operator==(ZFraction const& a, ZFraction const& b)
{
    if(a.estEgal(b))
        return true;
    else
        return false;
}

bool operator!=(ZFraction const& a, ZFraction const& b)
{
    if(a.estEgal(b))
        return false;
    else
        return true;
}

bool operator<(ZFraction const& a, ZFraction const& b) //Vrai si a<b
donc si a est plus petit que b
{
    if(a.estPlusPetitQue(b))
        return true;
    else
        return false;
}

bool operator>(ZFraction const& a, ZFraction const& b) //Vrai si a>b
donc si b est plus petit que a
{
    if(b.estPlusPetitQue(a))
        return true;
    else
        return false;
}

bool operator<=(ZFraction const& a, ZFraction const& b) //Vrai si
a<=b donc si b n'est pas plus petit que a
{
    if(b.estPlusPetitQue(a))
        return false;
    else
        return true;
}

bool operator>=(ZFraction const& a, ZFraction const& b) //Vrai si
a>=b donc si a n'est pas plus petit que b
{
    if(a.estPlusPetitQue(b))
        return false;
    else
        return true;
}

//Calcul du pgcd
int pgcd(int a, int b)
{
    while (b != 0)
    {
        const int t = b;
        b = a%b;
        a=t;
    }
    return a;
}
```

```
}
```

A vous maintenant de lire, tester et modifier ce code pour bien comprendre tout ce qui s'y passe. N'oubliez pas, la pratique est essentielle pour progresser en programmation.

Aller plus loin

Notre classe est terminée. Ou disons qu'elle remplit les conditions posées en début de chapitre. Mais vous en conviendrez, on est encore loin d'avoir fait le tour du sujet. On peut faire beaucoup plus avec des fractions. 😊

Je vous propose de télécharger le code source du TP si vous le souhaitez avant d'aller plus loin :

Télécharger le code source de zFraction

Voyons maintenant ce que l'on pourrait ajouter.

- **Ajouter des méthodes** `numérateur()` et `denominateur()` qui renvoient le numérateur et le dénominateur de la `ZFraction` sans la modifier.
- **Ajouter une méthode** `nombreReel()` qui convertit notre fraction en un `double`.
- **Simplifier les constructeurs** comme pour la classe `Duree`. En réfléchissant bien, on peut fusionner les trois constructeurs en un seul avec des valeurs par défaut.
- **Proposer plus d'opérateurs**. Nous avons implémenté l'addition et la multiplication. Il nous manque la soustraction et la division.
- Pour l'instant, notre classe ne gère que les fractions positives. Cela n'est pas suffisant ! Il faudrait **permettre des fractions négatives**.
Si vous vous lancez dans cette tâche, il va falloir faire des choix importants. La manière de gérer le signe par exemple. Ce que je vous propose c'est de toujours placer le signe de la fraction au numérateur. Ainsi, $\frac{1}{-4}$ devra être automatiquement converti en $-\frac{1}{4}$. En plus de simplifier les fractions, vos opérateurs devront donc aussi veiller à placer le signe au bon endroit.
A nouveau, je vous conseille d'utiliser une méthode privée.
- Si vous permettez l'utilisation de fractions négatives, alors il serait bien de **proposer l'opérateur "moins unaire"**. C'est l'opérateur qui transforme un nombre positif en nombre négatif comme dans `b = -a` ;. Je ne vous ai pas parlé de cet opérateur. Comme les autres opérateurs arithmétiques, il se déclare en-dehors de la classe. Son prototype est :

Code : C++

```
ZFraction operator-(ZFraction const& a);
```

C'est nouveau, mais pas très difficile si l'on utilise les bonnes méthodes de la classe. 😊

- **Ajouter des fonctions mathématiques** telles que `abs()`, `sqrt()`, `pow()` prenant en argument des `ZFraction`. Pensez à inclure l'en-tête `cmath`. 😊

Je pense que cela va vous demander pas mal de travail. 😊 Mais c'est tout bénéfice pour vous. Il faut pas mal d'expérience avec les classes pour arriver à "penser objet" et il n'y a que la pratique qui peut vous aider.

Je ne vais pas vous fournir une correction détaillée pour chacun de ces points. Mais je peux vous proposer une solution possible :

Télécharger le code source de zFraction avec les améliorations proposées

Si vous avez d'autres idées, n'hésitez pas à les ajouter à votre classe.

J'espère que tout c'est bien déroulé. Si ce n'est pas le cas, je vous invite à utiliser le [forum C++](#) pour y poser vos questions. Il y aura forcément quelqu'un qui saura vous répondre. 😊

Ce TP vous a, j'en suis sûr, aidé à bien saisir tout ce qui se cache derrière les classes. Vous avez notamment dû réfléchir au choix

des attributs, au choix des méthodes, aux constructeurs, aux opérateurs, ...

En somme, vous en savez déjà beaucoup ! Nous avons effectué un bon bout du chemin.

Dans les prochains chapitres, nous allons aborder des notions un peu plus complexes sur la POO, l'héritage et le polymorphisme notamment.

Classes et pointeurs

Dans les chapitres précédents, j'ai volontairement évité d'introduire les pointeurs avec les classes. En effet, les pointeurs en C++ sont un vaste sujet, et un sujet sensible. Comme vous l'avez probablement remarqué par le passé, bien gérer les pointeurs est essentiel car à la moindre erreur votre programme risque de :

- Consommer trop de mémoire parce que vous oubliez de libérer certains éléments
- Voire tout simplement de planter si votre pointeur pointe vers n'importe où dans la mémoire

Comment associe-t-on classes et pointeurs ? Quelles sont les règles à connaître, les bonnes habitudes à prendre ? Voilà un sujet qui méritait au moins un chapitre à lui tout seul. 😊



Attention : c'est un chapitre que je classe entre "très difficile" et "très très difficile". Bref, vous m'avez compris, les pointeurs en C++ c'est pas de la tarte, alors quadruplez d'attention lorsque vous lirez ce chapitre. Le sujet est complexe et épineux, je ne vous le dirai pas deux fois. 😬

Pointeur d'une classe vers une autre classe

Reprenons notre classe Personnage 😊

Dans les précédents chapitres, nous lui avons ajouté une Arme que nous avons directement intégré à ses attributs :

Code : C++

```
class Personnage
{
    public:

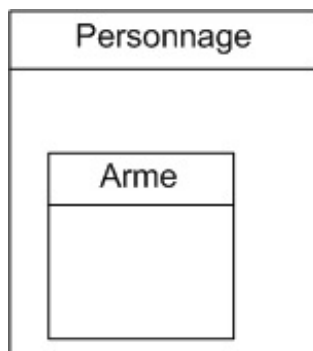
        // Quelques méthodes...

    private:

        Arme m_arme; // L'Arme est "contenue" dans le Personnage
        // ...
};
```

Il y a plusieurs façons différentes d'associer des classes entre elles. Celle-ci fonctionne bien dans notre cas, mais l'Arme est vraiment "liée" au Personnage. Elle ne peut pas en sortir.

Schématiquement, ça donnerait quelque chose de ce genre :



L'Arme est vraiment *dans* le Personnage.

Il y a une autre technique, plus souple, qui permet plus de possibilités, mais qui est plus complexe : ne pas intégrer l'Arme au Personnage et utiliser un pointeur à la place. Au niveau de la déclaration de la classe, le changement correspond à... une étoile en plus :

Code : C++

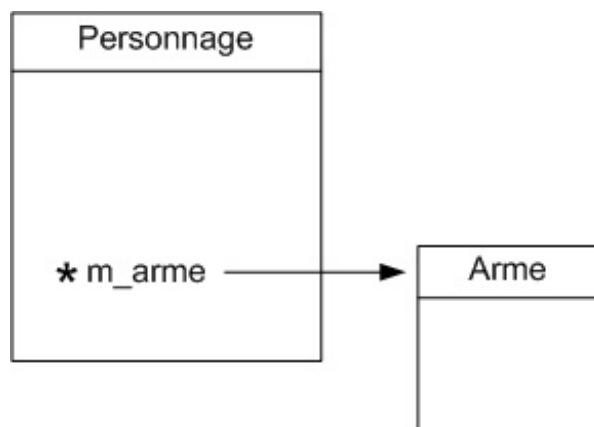
```
class Personnage
{
    public:

        // Quelques méthodes...

    private:

        Arme *m_arme; // L'Arme est un pointeur, l'objet n'est plus
        contenu dans le Personnage
        // ...
};
```

Notre Arme étant un pointeur, on ne peut plus dire qu'elle appartient au Personnage.
En schéma, ça donne ça :



On considère que l'arme est maintenant externe au personnage.
Les avantages de cette technique sont les suivants :

- Le Personnage peut changer d'Arme en faisant tout simplement pointer `m_arme` vers un autre objet. Par exemple, si le Personnage possède un inventaire (dans un sac à dos), il peut changer son Arme à tout moment en modifiant le pointeur.
- Le Personnage peut donner son Arme à un autre Personnage, il suffit de changer les pointeurs de chacun des personnages.
- Si le Personnage n'a plus d'Arme, il suffit de mettre le pointeur `m_arme` à 0.



Les pointeurs permettent de régler le problème que l'on avait vu pour le jeu de stratégie Warcraft III. Un personnage peut avoir une cible qui change grâce à un pointeur interne, exactement comme ici.

Mais des défauts, il y en a aussi. Gérer une classe qui contient des pointeurs, c'est pas de la tarte vous pouvez me croire, et d'ailleurs vous allez le voir. 🤪



Alors, faut-il utiliser un pointeur ou pas pour l'arme ? Les 2 façons de faire sont valables, et ont chacune leurs avantages et défauts. Utiliser un pointeur est probablement ce qu'il y a de plus souple, mais c'est aussi plus difficile. Retenez donc qu'il n'y a pas de "meilleure" méthode adaptée à tous les cas, ce sera à vous de choisir en fonction de votre cas si vous intégrez directement un objet dans une classe ou si vous utilisez un pointeur.

Gestion de l'allocation dynamique

On va ici voir comment on travaille quand une classe contient des pointeurs vers des objets.

On travaille là encore sur notre classe Personnage et je suppose que vous avez mis l'attribut `m_arme` en pointeur comme je l'ai montré un peu plus haut :

Code : C++

```

class Personnage
{
    public:

        // Quelques méthodes...

    private:

        Arme *m_arme; // L'Arme est un pointeur, l'objet n'est plus
        contenu dans le Personnage
        // ...
};

```

(je ne réécris volontairement pas tout le code, juste l'essentiel pour qu'on puisse se concentrer dessus)

Notre arme étant un pointeur, il va falloir faire une allocation dynamique avec **new** pour créer l'objet. Sinon, l'objet ne se créera pas tout seul. 😊

Allocation de mémoire pour l'objet

L'allocation de mémoire pour notre arme se fait où à votre avis ?

Il n'y a pas 36 endroits pour ça : c'est dans le **constructeur**. C'est en effet le rôle du constructeur que de faire en sorte que l'objet soit bien construit, donc notamment que tous les pointeurs pointent vers quelque chose. 😊

Dans notre cas, on est obligé de faire une allocation dynamique, donc d'utiliser **new**. Voici ce que ça donne dans le constructeur par défaut :

Code : C++

```

Personnage::Personnage() : m_arme(0), m_vie(100), m_mana(100)
{
    m_arme = new Arme();
}

```

Si vous vous souvenez bien, on avait aussi fait un second constructeur pour ceux qui veulent que le Personnage commence avec une arme plus puissante dès le départ. Il faut là aussi y faire une allocation dynamique :

Code : C++

```

Personnage::Personnage(string nomArme, int degatsArme) : m_arme(0),
m_vie(100), m_mana(100)
{
    m_arme = new Arme(nomArme, degatsArme);
}

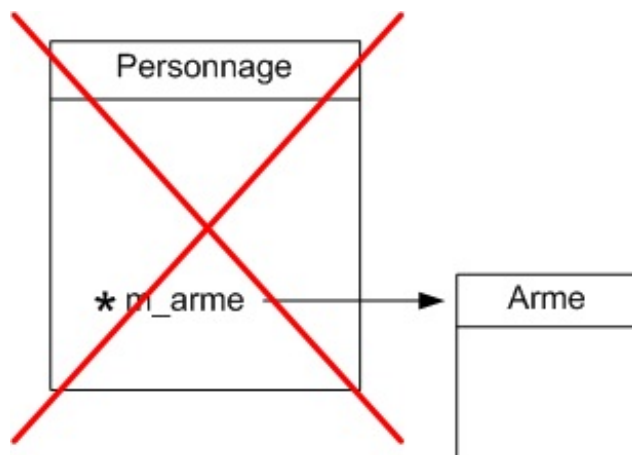
```

Explications : **new Arme()** appelle le constructeur par défaut de la classe **Arme**, tandis que **new Arme(nomArme, degatsArme)** appelle le constructeur surchargé. Le **new** renvoie l'adresse de l'objet créé, adresse qui est stockée dans notre pointeur **m_arme**.

On a d'abord initialisé le pointeur à 0 dans la liste d'initialisation par sécurité, puis on a fait l'allocation avec le **new** entre les accolades du constructeur.

Désallocation de mémoire pour l'objet

Notre arme étant un pointeur, lorsque l'objet de type Personnage est supprimé l'arme ne disparaît pas toute seule ! Si on fait juste un new dans le constructeur, et rien dans le destructeur, il va se passer ceci lorsque l'objet de type Personnage sera détruit :



L'objet de type Personnage va bel et bien disparaître, mais l'objet de type Arme va subsister en mémoire et il n'y aura plus aucun pointeur pour se "rappeler" de son adresse. En clair, l'arme va traîner en mémoire et on ne pourra plus jamais la supprimer. C'est ce qu'on appelle une *fuite de mémoire*.

Pour résoudre ce problème, il faut faire un delete de l'arme dans le **destructeur** du personnage afin que l'arme soit supprimée *avant* le personnage. Le code est tout simple :

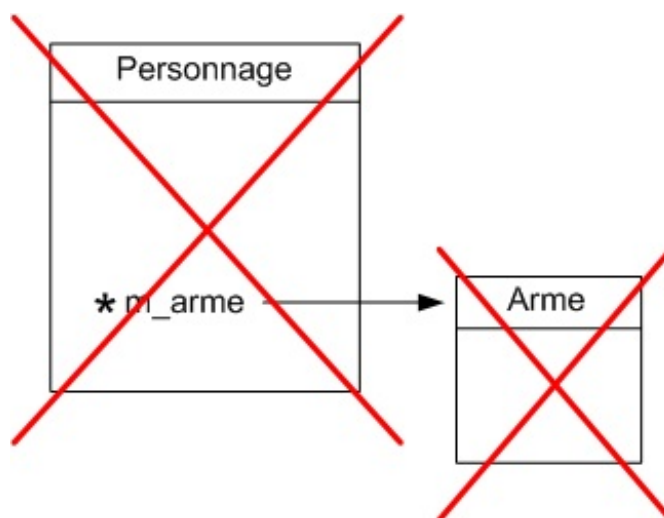
Code : C++

```
Personnage::~~Personnage()
{
    delete m_arme;
}
```

Cette fois le destructeur est réellement indispensable. Maintenant, lorsque quelqu'un demandera à détruire le Personnage, il va se passer ceci :

1. Appel du destructeur... et donc dans notre cas suppression de l'Arme (avec le delete).
2. Puis enfin suppression du Personnage.

Au final, les 2 objets seront bel et bien supprimés et la mémoire sera propre :



N'oubliez pas que `m_arme` est maintenant un pointeur !

Cela implique de changer toutes les méthodes qui l'utilisent. Par exemple :

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.getDegats());
}
```

... devient :

Code : C++

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme->getDegats());
}
```

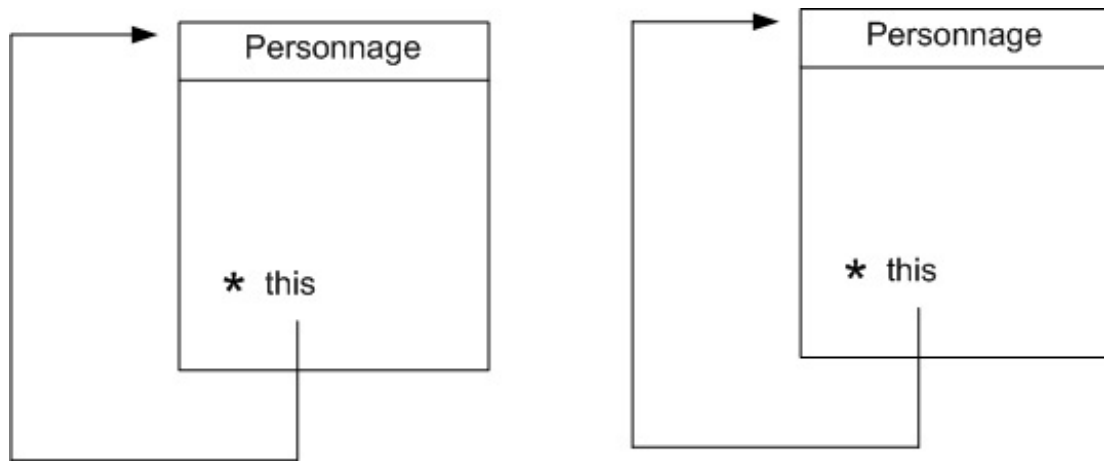
Notez la différence : le point a été remplacé par la flèche, car `m_arme` est un pointeur.

Le pointeur **this**

Ce chapitre étant difficile, je vous propose un passage un peu plus *cool*. Puisqu'on parle de POO et de pointeurs, je me dois de vous parler du pointeur **this**.

Pas de panique, c'est très simple, ça ira vite et vous ne sentirez aucune douleur. 🤪

Dans toutes les classes, on dispose d'un pointeur ayant pour nom *this*. Ce pointeur pointe **vers l'objet actuel**. Je reconnais que ce n'est pas simple à imaginer, mais je pense que ça passera mieux avec un schéma maison :



Chaque objet (ici de type `Personnage`) possède un pointeur `this` qui pointe vers... l'objet lui-même !



`this` étant utilisé par le langage C++ dans toutes les classes, vous ne pouvez donc pas créer de variable appelée `this` car cela créerait un conflit. De même, si vous commencez à essayer d'appeler vos variables `class`, `new`, `delete`, `return`, etc. forcément ça risque de coïncider un peu. 😊

Ces mots-clés sont ce qu'on appelle des mots-clés réservés. Le langage C++ se les réserve pour son usage personnel, vous n'avez donc pas le droit de créer des variables (ou des fonctions) portant ces noms-là.



Mais... à quoi peut bien servir `this` ???

Répondre à cette question me sera délicat. 😬

Je peux vous donner un exemple : vous êtes dans une méthode de votre classe, et cette méthode doit renvoyer un pointeur vers l'objet auquel elle appartient. Sans le `this`, on ne pourrait pas l'écrire. Voilà ce que ça pourrait donner :

Code : C++

```
Personnage* Personnage::getAdresse() const
{
    return this;
}
```

Mais nous l'avons déjà rencontré une fois. Lors de la surcharge de l'opérateur `+=`. Souvenez-vous, notre opérateur ressemblait à cela :

Code : C++


```
Duree& Duree::operator+=(const Duree &duree2)
{
    //Des calculs compliqués...

    return *this;
}
```

`this` étant un pointeur sur un objet, `*this` est l'objet lui-même ! Notre opérateur renvoie donc l'objet lui-même en retour. La raison pour laquelle on doit renvoyer l'objet est compliquée, mais c'est la forme correcte des opérateurs. Je vous propose donc de simplement apprendre cette syntaxe par cœur.

À part pour la surcharge des opérateurs, vous n'avez certainement pas à utiliser `this` dans l'immédiat mais il arrivera un jour où, pour résoudre un problème particulier, vous aurez besoin d'un tel pointeur. Ce jour-là, souvenez-vous qu'un objet peut

"retrouver" son adresse à l'aide du pointeur **this**.

Comme c'est l'endroit le plus adapté pour en parler dans ce cours, j'en profite. Ça ne va pas changer votre vie tout de suite, mais il se peut que bien plus tard, dans plusieurs chapitres je vous dise tel un vieillard sur sa canne "Souvenez-vous, souvenez-vous du pointeur **this** ! ". Alors ne l'oubliez pas !

Le constructeur de copie

Le **constructeur de copie** est une *surcharge* particulière du constructeur.

Le constructeur de copie devient généralement indispensable dans une classe qui contient des pointeurs, et ça tombe bien vu que c'est justement notre cas ici. 😊

Le problème

Pour bien comprendre l'intérêt du constructeur de copie, voyons voir concrètement ce qui se passe lorsqu'on crée un objet en l'affectant par... un autre objet ! Par exemple :

Code : C++

```
int main()
{
    Personnage goliath("Epée aiguisée", 20);

    Personnage david(goliath); // On crée david à partir de
    goliath. David sera une "copie" de goliath.

    return 0;
}
```

Lorsqu'on construit un objet en lui affectant directement un autre objet, comme on vient de le faire ici, le compilateur appelle une méthode appelée constructeur de copie.

Le rôle du constructeur de copie est de copier la valeur de tous les attributs du premier objet dans le second. Donc david récupère la vie de goliath, la mana de goliath, etc.



Dans quels cas le constructeur de copie est-il appelé ?

On vient de le voir, le constructeur de copie est appelé lorsqu'on crée un nouvel objet en l'affectant par la valeur d'un autre :

Code : C++

```
Personnage david(goliath); // Appel du constructeur de copie (cas 1)
```

Ceci est strictement équivalent à écrire :

Code : C++

```
Personnage david = goliath; // Appel du constructeur de copie (cas 2)
```

Dans ce second cas le constructeur de copie est là aussi appelé.

Mais ce n'est pas tout ! Lorsque vous envoyez un objet à une fonction sans utiliser de pointeur ni de référence, l'objet est là aussi copié !

Imaginons la fonction :

Code : C++

```
void maFonction(Personnage unPersonnage)
{
}

```

Si vous appelez cette fonction qui n'utilise pas de pointeur ni de référence, alors l'objet sera copié en utilisant un constructeur de copie au moment de l'appel de la fonction :

Code : C++

```
maFonction(Goliath); // Appel du constructeur de copie (cas 3)

```

Bien entendu, il est préférable d'utiliser une référence en général car l'objet n'a pas besoin d'être copié, donc ça va bien plus vite et ça prend moins de mémoire. Toutefois, il arrivera des cas où vous aurez besoin de créer une fonction comme ici qui fait une copie de l'objet.



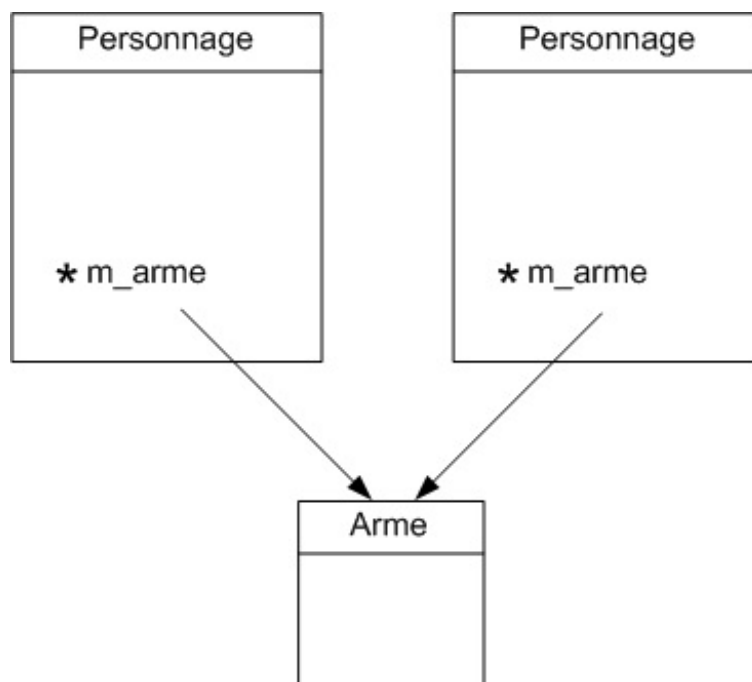
Si vous n'écrivez pas vous-mêmes un constructeur de copie pour votre classe, il sera généré automatiquement pour vous par le compilateur. Ok, c'est sympa de sa part, mais le compilateur est... comment dire pour pas le froisser... bête.



En fait, le constructeur de copie généré se contente de copier la valeur de tous les attributs... même des pointeurs !

Le problème ? Eh bien justement, il se trouve qu'un des attributs est un pointeur dans notre classe Personnage ! Que fait l'ordinateur ? Il copie la valeur du pointeur, donc l'adresse de l'arme. Au final, les 2 objets ont un pointeur qui pointe vers le même objet de type Arme !

Ah les fourbes !



L'ordinateur a copié le pointeur, et donc les 2 pointeurs pointent vers la même arme !



Si on ne fait rien pour régler ça, imaginez ce qu'il va se passer lorsque les 2 personnages seront détruits... Le premier sera détruit, ainsi que son arme car le destructeur ordonnera la suppression de l'arme avec un **delete**. Et quand



arrivera le tour du second personnage, le **delete** va planter (et votre programme avec 😬) parce que l'arme aura *déjà* été détruite !

Le constructeur de copie généré automatiquement par le compilateur n'est pas assez intelligent pour comprendre qu'il faut allouer de la mémoire pour une autre arme... Qu'à cela ne tienne, nous allons le lui expliquer. 😬

Création du constructeur de copie

Le constructeur de copie, comme je vous l'ai dit un peu plus haut, est une surcharge particulière du constructeur. C'est un constructeur qui prend pour paramètre... une référence constante vers un objet du même type !

Si vous ne trouvez pas ça clair, peut-être qu'un exemple vous aidera. 😬

Code : C++

```
class Personnage
{
    public:

        Personnage();
        Personnage(Personnage const& personnageACopier); // Le prototype du
        constructeur de copie
        Personnage(std::string nomArme, int degatsArme);
        ~Personnage();

        /*
        ... plein d'autres méthodes qui ne nous intéressent pas ici
        */

    private:

        int m_vie;
        int m_manana;
        Arme *m_arme;
};
```

En résumé, le prototype d'un constructeur de copie est :

Code : C++

```
Objet(Objet const& objetACopier);
```

Le **const** indique juste qu'on n'a pas le droit de modifier les valeurs de l'objetACopier (c'est logique, on a juste besoin de "lire" ses valeurs pour le copier).

Écrivons l'implémentation de ce constructeur. Il va falloir copier tous les attributs du personnageACopier dans le personnage actuel. Commençons par les attributs "simples", c'est-à-dire ceux qui ne sont pas des pointeurs :

Code : C++

```
Personnage::Personnage(Personnage const& personnageACopier)
    : m_vie(personnageACopier.m_vie),
      m_manana(personnageACopier.m_manana), m_arme(0)
{
}
```

Vous vous demandez peut-être comment cela se fait qu'on puisse accéder aux attributs `m_vie` et `m_mana` du `personnageACopier` ? Si vous vous l'êtes demandé, je vous félicite, ça veut dire que le principe d'encapsulation commence à rentrer dans votre tête. 😊



Eh oui, en effet, `m_vie` et `m_mana` sont privés, donc on ne peut pas y accéder depuis l'extérieur de la classe... sauf qu'il y a une exception ici : on est dans une méthode de la classe `Personnage`, et on a le droit d'accéder à tous les éléments (même privés) d'un autre `Personnage`.

C'est un peu tordu je l'avoue, mais dans le cas présent ça nous simplifie grandement la vie. Retenez donc qu'un objet de type `X` peut accéder à tous les éléments (même privés) d'un autre objet s'il est du même type `X`.

Il reste maintenant à "copier" `m_arme`. Si on écrit :

Code : C++

```
m_arme = personnageACopier.m_arme;
```

... on fait exactement la même erreur que le compilateur, c'est-à-dire qu'on ne copie que l'adresse de l'objet de type `Arme`, et pas l'objet en entier !

Pour résoudre le problème, il va falloir copier l'objet de type `Arme` en faisant une allocation dynamique, donc un **new**. Attention, accrochez-vous parce que ce n'est pas simple. 😬

Si on fait :

Code : C++

```
m_arme = new Arme();
```

... on va bien créer une nouvelle arme, mais on utilisera le constructeur par défaut, donc cela créera l'arme de base. Or, on veut avoir exactement la même arme que celle du `personnageACopier` (ben oui, c'est un constructeur de copie 😊).

La bonne nouvelle, comme je vous l'ai dit plus haut, c'est que le constructeur de copie est automatiquement généré par le compilateur. Tant que la classe n'utilise pas de pointeurs vers des attributs, il n'y a pas de danger. Et ça tombe bien, la classe `Arme` n'utilise pas de pointeurs, on va donc pouvoir se contenter du constructeur qui a été généré.

Il faut donc appeler le constructeur de copie de `Arme`, en envoyant en paramètre l'objet à copier. Vous pourriez penser qu'il faut faire ceci :

Code : C++

```
m_arme = new Arme(personnageACopier.m_arme);
```

Presque ! Sauf que `m_arme` est un pointeur, et le prototype du constructeur de copie est :

Code : C++

```
Arme(Arme const& arme);
```

... ce qui veut dire qu'il faut envoyer l'objet lui-même et pas son adresse. Vous vous souvenez comment on fait pour obtenir

l'objet (ou la variable) à partir de son adresse ? On utilise l'étoile * !
Ce qui donne au final :

Code : C++

```
m_arme = new Arme (*(personnageACopier.m_arme));
```

Cette ligne alloue dynamiquement une nouvelle arme, en se basant sur l'arme du `personnageACopier`. Pas simple je le reconnais, mais relisez plusieurs fois les étapes de mon raisonnement et vous allez comprendre. 🤔

Pour bien suivre tout ce que j'ai dit, il faut vraiment que vous soyez au point sur tout : les pointeurs, les références, et les... constructeurs de copie. 🤔

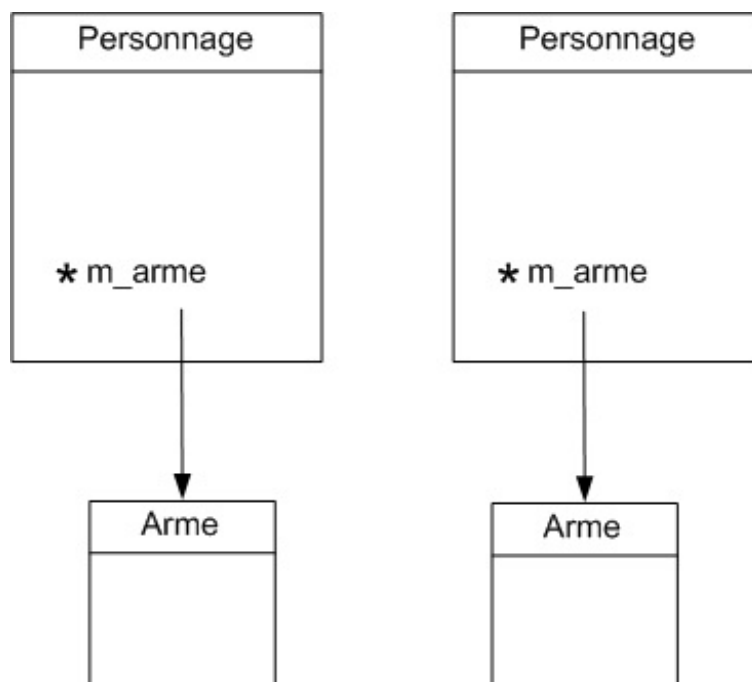
Le constructeur de copie une fois terminé

Le bon constructeur de copie ressemblera donc à ceci au final :

Code : C++

```
Personnage::Personnage(Personnage const& personnageACopier)
: m_vie(personnageACopier.m_vie),
mMana(personnageACopier.mMana), m_arme(0)
{
    m_arme = new Arme (*(personnageACopier.m_arme));
}
```

Ainsi, nos 2 personnages ont tous deux une arme identique, mais dupliquée afin d'éviter les problèmes que je vous ai expliqués plus haut :



L'opérateur d'affectation

Nous avons déjà parlé de la surcharge des opérateurs. Mais il y en a un que je ne vous ai pas présenté. Il s'agit de l'opérateur

d'affectation (**operator=**).



Le compilateur écrit un opérateur d'affectation par défaut automatiquement, mais c'est un opérateur "bête". Cet opérateur bête se contente de copier les valeurs des attributs un à un dans le nouvel objet. Comme pour le constructeur de copie généré par le compilateur.

La méthode **operator=** sera appelée dès qu'on essaie d'affecter une valeur à notre objet. C'est le cas par exemple si on affecte à notre objet la valeur d'un autre objet :

Code : C++

```
david = goliath;
```

Ne confondez pas le constructeur de copie avec la surcharge de l'opérateur = (**operator=**). Ils se ressemblent beaucoup, mais il y a une différence : le constructeur de copie est appelé lors de l'initialisation (à la création de l'objet) tandis que la méthode **operator=** est appelée si on essaie d'affecter un autre objet par la suite, après son initialisation.



Code : C++

```
Personnage david = goliath; // Constructeur de copie
david = goliath; // operator=
```

Cette méthode effectue le même travail que le constructeur de copie. Écrire son implémentation est donc relativement simple. Une fois qu'on a compris le principe bien sûr. 😊

Code : C++

```
Personnage& Personnage::operator=(Personnage const&
personnageACopier)
{
    if(this != &personnageACopier) //On vérifie que notre objet n'est
    pas le même que celui reçu en argument
    {
        m_vie = personnageACopier.m_vie; //On copie tous les
        champs
        m_mana = personnageACopier.m_mana;
        delete m_arme;
        m_arme = new Arme(* (personnageACopier.m_arme));
    }

    return *this; //On renvoie l'objet lui-même
}
```

Il y a tout de même quatre différences :

- Comme ce n'est pas un constructeur, on ne peut pas utiliser la liste d'initialisation et donc tout se passe entre les accolades.
- Il faut penser à vérifier que l'on n'est pas en train de faire `david=david`. On doit donc vérifier que l'on travaille avec deux objets distincts. Il faut donc vérifier que leurs adresses mémoires (**this** et `&personnageACopier`) sont différentes.
- Il faut renvoyer ***this** comme pour les opérateurs `+=`, `-=`, etc. C'est une règle à respecter.
- Il faut penser à supprimer l'ancienne arme avant de créer la nouvelle. C'est ce qui est fait à l'instruction **delete** surlignée du code. Ceci n'était pas nécessaire dans le constructeur de copie puisque le personnage ne possédait pas d'arme avant.



Cet opérateur est toujours similaire à celui que je vous donne pour la classe `Personnage`. Les seuls éléments qui changent d'une classe à l'autre sont les lignes qui se trouvent dans le `i.f`. Je vous ai en quelque sorte donné la recette universelle. 🤖

Il y a une chose importante à retenir au sujet de cet opérateur : il va toujours de paire avec le constructeur de copie.

Si l'on a besoin d'écrire un constructeur de copie, alors il faut aussi obligatoirement écrire une surcharge de `operator=`.

C'est une règle très importante à respecter. Vous risquez de graves problèmes de pointeurs si vous ne la respectez pas.

La POO n'est pas simple comme vous commencez à vous en rendre compte, surtout quand on commence à manipuler des objets avec des pointeurs. Heureusement, vous aurez l'occasion de pratiquer tout cela par la suite, et vous allez petit à petit prendre l'habitude d'éviter les pièges des pointeurs.

Si vous êtes en train de vous shooter à l'aspirine pour éviter que votre tête n'explode, je vous conseille de conserver encore des munitions 😊

En effet, on n'a pas fini d'en découdre avec la POO et il vous reste encore beaucoup de choses à apprendre. Heureusement, enfin si ça peut vous rassurer, ce chapitre était probablement l'un des plus difficiles de tout le cours (mais pas nécessairement LE plus difficile 🤖).

Sachez quoiqu'il en soit que les pointeurs en C++ sont de véritables casse-têtes, même pour les programmeurs plus expérimentés. Il faut faire constamment attention, car une fuite de mémoire (oubli de libérer des objets) est très vite arrivée, et je ne vous parle pas des plantages de programme que ça peut occasionner. Une très très grande part des plantages des programmes que vous connaissez sont dûs à une mauvaise gestion de la mémoire, c'est vous dire !

Dans le prochain chapitre, nous nous rapprocherons d'un des thèmes majeurs de la programmation orientée objet, quelque chose d'indispensable à quoi vous ne pouvez échapper et qui porte un bien funeste nom : **l'héritage**. 🤖

Qu'on ne s'y trompe pas : tout ceci est peut-être complexe et pas toujours très "amusant" à apprendre, mais vous en aurez vraiment besoin dans la partie III lorsque nous travaillerons avec la librairie Qt pour créer des fenêtres, travailler en réseau, etc. Donc on se motive, et on continue ! 😊

L'héritage

Nous allons maintenant découvrir une des notions les plus importantes de la POO : **l'héritage**.

Qu'on se rassure, il n'y aura pas de morts.

(voilà ça c'est fait)

L'héritage, c'est un concept très important qui fait à lui tout seul peut-être plus de la moitié de l'intérêt de la programmation orientée objet. Bref, ça rigole pas. C'est pas le moment de s'endormir au fond, je vous ai à l'œil. 🤖

Nous allons dans ce chapitre réutiliser notre exemple de la classe Personnage, mais on va beaucoup le simplifier pour se concentrer uniquement sur ce qui est important. En clair, on va juste garder le strict minimum, histoire d'avoir un exemple simple mais que vous connaissez déjà.

Allez, bon courage, cette notion n'est pas bien dure à comprendre, elle est juste très riche.

Exemple d'héritage simple

"Héritage", c'est un drôle de mot pour de la programmation hein 🤖

Alors c'est quoi ? C'est une technique qui permet de créer une classe à partir d'une autre classe. Elle lui sert de modèle, de base de départ. Cela permet d'éviter à avoir à réécrire un même code source plusieurs fois.

Comment reconnaître un héritage ?

C'est LA question à se poser. Certains ont tellement été traumatisés par l'héritage qu'ils en voient partout, d'autres au contraire (surtout les débutants) se demandent à chaque fois s'il y a un héritage à faire ou pas. Pourtant, ce n'est pas "mystique", il est très facile de savoir s'il y a une relation d'héritage entre 2 classes.

Comment ? En suivant cette règle très simple :

Il y a héritage quand on peut dire :
"A est un B"

Pas de panique c'est pas des maths 🤖

Prenez un exemple très simple. On peut dire "Un guerrier est un personnage", ou encore "Un magicien est un personnage". Donc on peut faire un héritage : "La classe Guerrier hérite de Personnage", "La classe Magicien hérite de Personnage".

Pour vous imprégner, voici quelques autres bons exemples où un héritage peut être fait :

- Une voiture est un véhicule (Voiture hérite de Vehicule)
- Un bus est un véhicule (Bus hérite de véhicule)
- Un moineau est un oiseau (Moineau hérite d'Oiseau)
- Un corbeau est un oiseau (Corbeau hérite d'Oiseau)
- Un chirurgien est un docteur (Chirurgien hérite de Docteur)
- Un diplodocus est un dinosaure (Diplodocus hérite de Dinosaur)
- etc.

En revanche, vous ne pouvez pas dire "Un dinosaure est un diplodocus", ou encore "Un bus est un oiseau". Donc on ne peut pas faire d'héritage dans ces cas-là, du moins ça n'aurait aucun sens 🤖



J'insiste, mais il est très important de respecter cette règle. Vous risquez de vous retrouver avec des gros problèmes de logique dans vos codes si vous ne le faites pas.

Nous allons voir comment réaliser un héritage en C++, mais d'abord il faut que je pose l'exemple sur lequel on va travailler 🤖

Notre exemple : la classe Personnage

Petit rappel : cette classe représente un personnage d'un jeu vidéo de type RPG (jeu de rôle). Il n'est pas nécessaire de savoir jouer ou d'avoir joué à un RPG pour suivre mon exemple. J'ai juste choisi celui-là car il est plus ludique que la plupart des exemples barbaques que les profs d'informatique aiment utiliser (Voiture, Bibliothèque, Université, PompeAEssence...).

On va un peu simplifier notre classe Personnage. Voici ce sur quoi je vous propose de partir :

Code : C++ - Personnage.h

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include <iostream>
#include <string>

class Personnage
{
public:
    Personnage();
    void recevoirDegats(int degats);
    void coupDePoing(Personnage &cible) const;

private:
    int m_vie;
    std::string m_nom;
};

#endif
```

Notre Personnage a un nom et une quantité de vie.

On n'a mis qu'un seul constructeur, le constructeur par défaut. Il permet d'initialiser le Personnage avec un nom et lui donnera 100 points de vie.

Le Personnage peut recevoir des dégâts, via la méthode `recevoirDegats()` et en distribuer, via la méthode `coupDePoing()`.

A titre informatif, voici l'implémentation des méthodes dans `Personnage.cpp` :

Code : C++ - Personnage.cpp

```
#include "Personnage.h"

using namespace std;

Personnage::Personnage() : m_vie(100), m_nom("Jack")
{
}

void Personnage::recevoirDegats(int degats)
{
    m_vie -= degats;
}

void Personnage::coupDePoing(Personnage &cible) const
{
    cible.recevoirDegats(10);
}
```

Rien d'extraordinaire pour le moment.

La classe Guerrier hérite de la classe Personnage

Intéressons-nous maintenant à l'héritage. L'idée, c'est de créer une nouvelle classe qui est une sous-classe de Personnage. On dit que cette classe va *hériter* de Personnage.

Pour cet exemple, je vais créer une classe Guerrier qui hérite de Personnage. La définition de la classe, dans Guerrier.h, ressemble à ceci :

Code : C++ - Guerrier.h

```
#ifndef DEF_GUERRIER
#define DEF_GUERRIER

#include <iostream>
#include <string>
#include "Personnage.h" // Ne pas oublier d'inclure Personnage.h
pour pouvoir en hériter !

class Guerrier : public Personnage // Signifie : créer une classe
Guerrier qui hérite de la classe Personnage
{

};

#endif
```

Grâce à ce qu'on vient de faire, la classe Guerrier contiendra de base tous les attributs et toutes les méthodes de la classe Personnage.

Dans un tel cas, la classe Personnage est appelée la classe "Mère", et la classe Guerrier la classe "Fille".



Mais quel intérêt de créer une nouvelle classe si c'est pour qu'elle contienne les mêmes attributs et les mêmes méthodes ?

Attendez, justement ! Le truc, c'est qu'on peut rajouter des attributs et des méthodes spéciales dans la classe Guerrier. Par exemple, on pourrait rajouter une méthode qui ne concerne que les guerriers, du genre frapperCommeUnSourdAvecUnMarteau (bon ok c'est un nom de méthode un peu long j'avoue 🤔).

Code : C++ - Guerrier.h

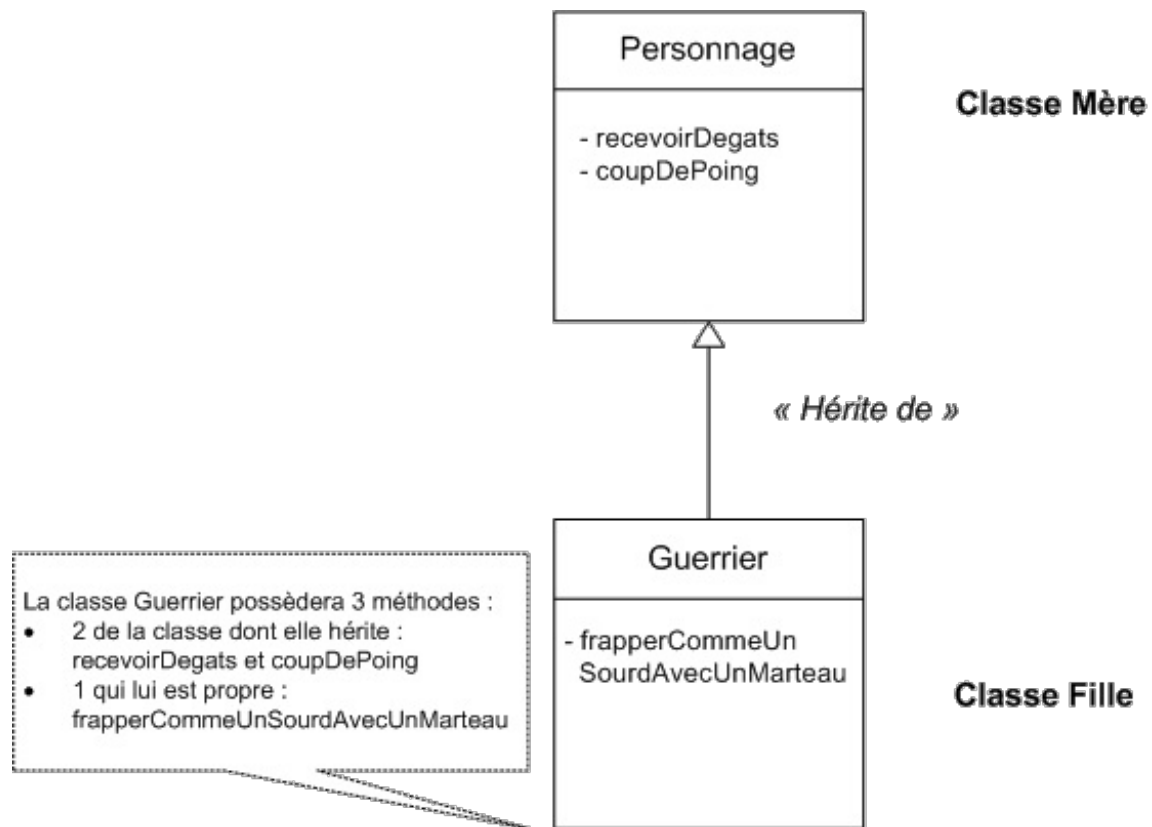
```
#ifndef DEF_GUERRIER
#define DEF_GUERRIER

#include <iostream>
#include <string>
#include "Personnage.h"

class Guerrier : public Personnage
{
    public:
        void frapperCommeUnSourdAvecUnMarteau() const; // Méthode
        qui ne concerne que les guerriers
};

#endif
```

Schématiquement, on représente la situation comme ça :



Le schéma se lit de bas en haut, c'est-à-dire "Guerrier hérite de Personnage".

Guerrier est la classe fille, Personnage est la classe mère. On dit que Guerrier est une "spécialisation" de la classe Personnage. Elle possède toutes les caractéristiques d'un Personnage (de la vie, un nom, elle peut recevoir des dégâts), mais possède en plus des caractéristiques propres au Guerrier comme `frapperCommeUnSourdAvecUnMarteau()`. 🤪



Retenez bien que lorsqu'on fait un héritage, on hérite des méthodes *et* des attributs.

Je n'ai pas représenté les attributs sur le schéma ci-dessus pour ne pas surcharger, mais la vie et le nom du Personnage sont bel et bien hérités, ce qui fait qu'un Guerrier possède aussi de la vie et un nom !

Vous commencez à comprendre le principe ? En C++, quand on a deux classes qui sont liées par la relation EST-UN, on utilise l'héritage pour mettre en évidence ce lien. Un Guerrier EST-UN Personnage amélioré qui possède une méthode supplémentaire.

Ce concept a l'air de rien comme ça, mais croyez-moi ça fait la différence ! Vous n'allez pas tarder à voir tout ce que ça a de puissant lorsque vous pratiquerez plus loin dans le cours.

La classe Magicien hérite aussi de Personnage

Tant qu'il n'y a qu'un seul héritage, l'intérêt semble encore limité. Mais multiplions un peu les héritages et les spécialisations et nous allons vite voir tout l'intérêt de la chose.

Par exemple, si on créait une classe Magicien qui va elle aussi hériter de Personnage ? Après tout, un Magicien est un Personnage, donc il peut récupérer les mêmes propriétés de base : de la vie, un nom, donner un coup de poing, etc.

La différence, c'est que le Magicien peut aussi envoyer des sorts magiques, par exemple `bouleDeFeu` et `bouleDeGlace`. Pour utiliser sa magie, il a une réserve de magie qu'on appelle "Mana" (ça va faire un attribut à rajouter). Quand la Mana tombe à zéro, il ne peut plus lancer de sort.

Code : C++ - Magicien.h

```

#ifndef DEF_MAGICIEN
#define DEF_MAGICIEN

#include <iostream>
#include <string>
  
```

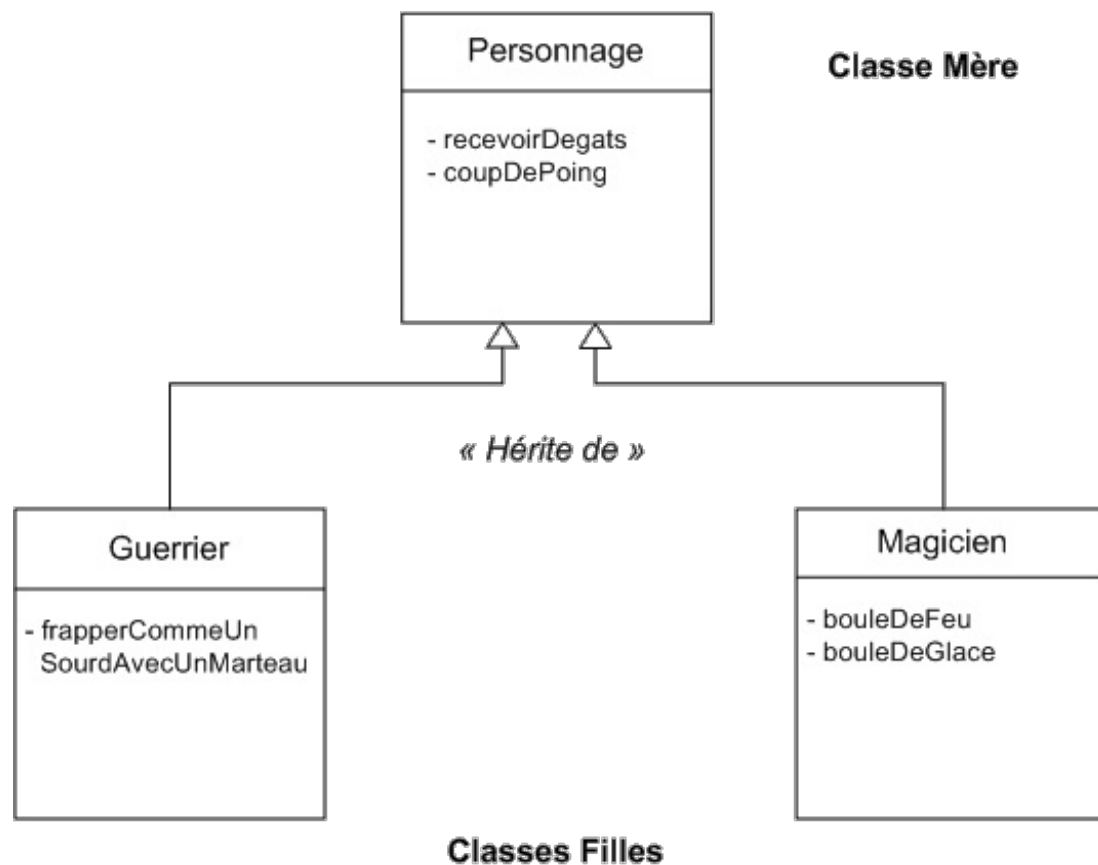
```
#include "Personnage.h"

class Magicien : public Personnage
{
    public:
        void bouleDeFeu() const;
        void bouleDeGlace() const;

    private:
        int m_manana;
};

#endif
```

Je ne vous donne pas l'implémentation des méthodes (le .cpp) ici, je veux juste que vous compreniez et reteniez le principe :

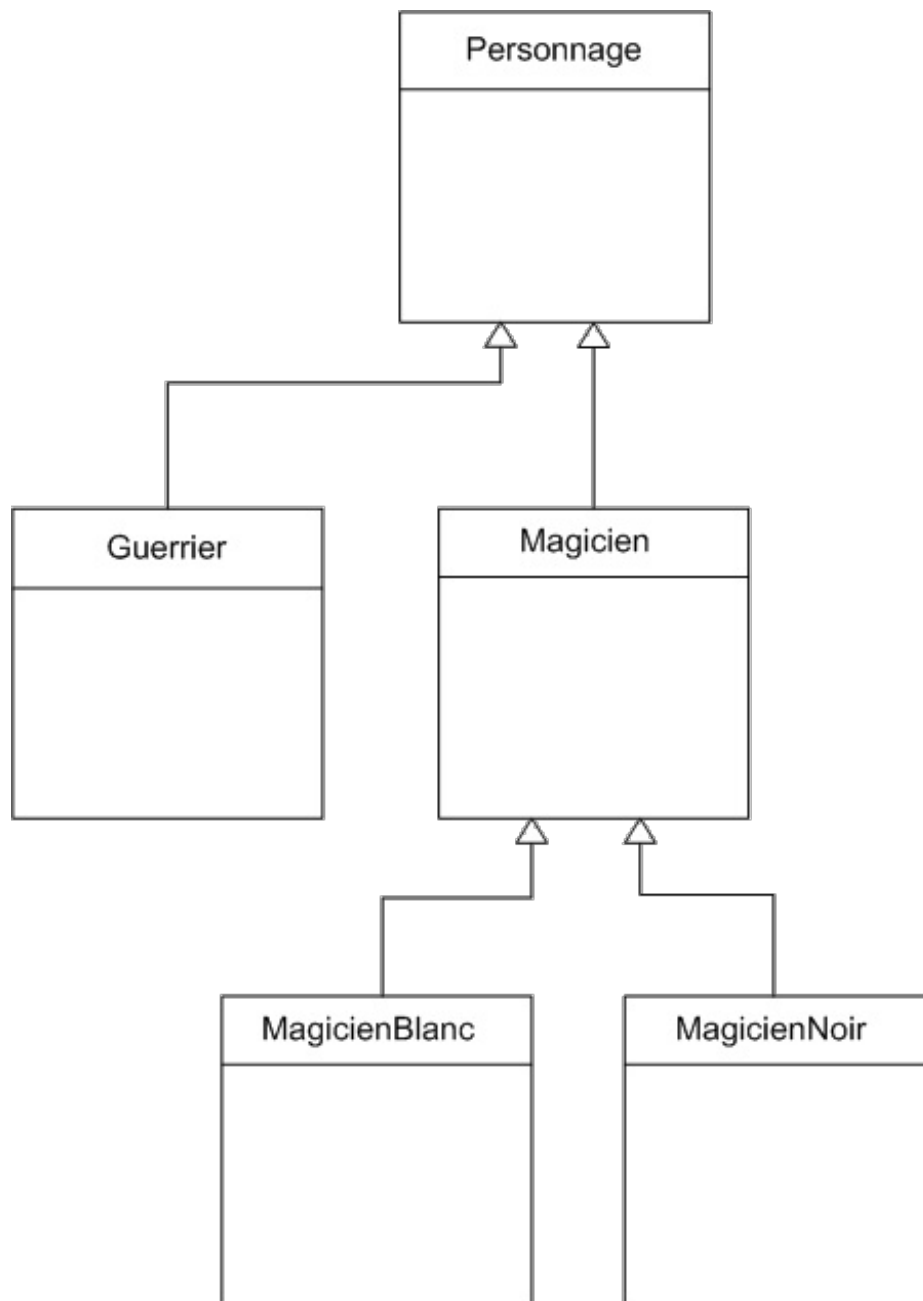


Notez que sur le schéma je n'ai représenté que les méthodes des classes, mais les attributs (vie, nom...) sont eux aussi hérités !

Et le plus beau, c'est qu'on peut faire une classe qui hérite d'une classe qui hérite d'une autre classe ! 😊

Imaginons qu'il y ait 2 types de magiciens : les magiciens blancs, qui sont des gentils qui envoient des sorts de guérison tout ça, et les magiciens noirs qui sont des méchants qui utilisent leurs sorts pour tuer des gens (super exemple, j'en suis fier).

Avada Kedavra !



Et ça pourrait continuer longtemps comme ça. Vous verrez dans la prochaine partie sur la bibliothèque C++ Qt qu'il y a souvent 5 ou 6 héritages qui sont faits à la suite. C'est vous dire si c'est utilisé !

La dérivation de type

Imaginons le code suivant :

Code : C++

```
Personnage monPersonnage;  
Guerrier monGuerrier;  
  
monPersonnage.coupDePoing(monGuerrier);  
monGuerrier.coupDePoing(monPersonnage);
```

Compilez : ça marche. Mais si vous êtes attentif, vous devriez vous demander *pourquoi* ça a marché, parce que normalement ça n'aurait pas dû !

... non, vous ne voyez pas ? 🤔

Allez un effort, voici le prototype de coupDePoing (il est le même dans la classe Personnage et dans la classe Guerrier rappelez-

vous) :

Code : C++

```
void coupDePoing(Personnage &cible) const;
```

Quand on fait `monGuerrier.coupDePoing(monPersonnage);`, on envoie bien un `Personnage` en paramètre. Mais quand on fait `monPersonnage.coupDePoing(monGuerrier);`, ça marche aussi et le compilateur ne hurle pas à la mort alors que, selon toute logique, il devrait ! En effet, la méthode `coupDePoing` attend un `Personnage` et on lui envoie un `Guerrier`. Pourquoi diable cela fonctionne-t-il ? 🤔

Eh bien... c'est justement une propriété très intéressante de l'héritage en C++ que vous venez de découvrir là. **On peut substituer un objet fille à un pointeur ou une référence d'un objet mère.** Ce qui veut dire, dans une autre langue que le chinois, qu'on peut faire ça :

Code : C++

```
Personnage *monPersonnage(0);  
Guerrier *monGuerrier = new Guerrier();  
  
monPersonnage = monGuerrier; // Mais... mais... Ca marche !?
```

Les 2 premières lignes n'ont rien d'extraordinaire : on crée un pointeur `Personnage` mis à 0, et un pointeur `Guerrier` qu'on initialise avec l'adresse d'un nouvel objet de type `Guerrier`.

Par contre, la dernière ligne est assez surprenante. Normalement, on ne *devrait pas* pouvoir donner à un pointeur de type `Personnage` un pointeur de type `Guerrier`. C'est comme mélanger des carottes et des patates, ça ne se fait pas.

Alors oui, en temps normal le compilateur n'accepte pas d'échanger des pointeurs (ou des références) de types différents. Or, `Personnage` et `Guerrier` ne sont pas n'importe quels types : `Guerrier` hérite de `Personnage`. Et la règle à connaître, c'est justement qu'on peut affecter un élément enfant à un élément parent ! En fait c'est logique puisque `Guerrier` EST UN `Personnage`. 😊



L'inverse est faux par contre ! On ne peut PAS faire :

`monGuerrier = monPersonnage;`

Ceci plante et est strictement interdit. Attention au sens de l'affectation donc.

Cela nous permet donc de placer un élément dans un pointeur (ou une référence) de type plus général. C'est très pratique dans notre cas lorsqu'on passe une cible en paramètre :

Code : C++

```
void coupDePoing(Personnage &cible) const;
```

Notre méthode `coupDePoing` est capable de faire mal à n'importe quel `Personnage` ! Qu'il soit `Guerrier`, `Magicien`, `MagicienBlanc`, `MagicienNoir` ou autre, c'est un `Personnage` après tout, donc on peut lui donner un `coupDePoing` 😊

C'est un peu choquant au début je le reconnais, mais on se rend compte au final qu'en fait c'est très bien fait. **Ca fonctionne, puisque la méthode `coupDePoing` ne fait qu'appeler des méthodes de la classe `Personnage` (`recevoirDegats`), et que ces méthodes se trouvent forcément dans toutes les classes filles (`Guerrier`, `Magicien`).**

Relisez-moi, essayez de comprendre, vous devriez saisir pourquoi ça marche 😊



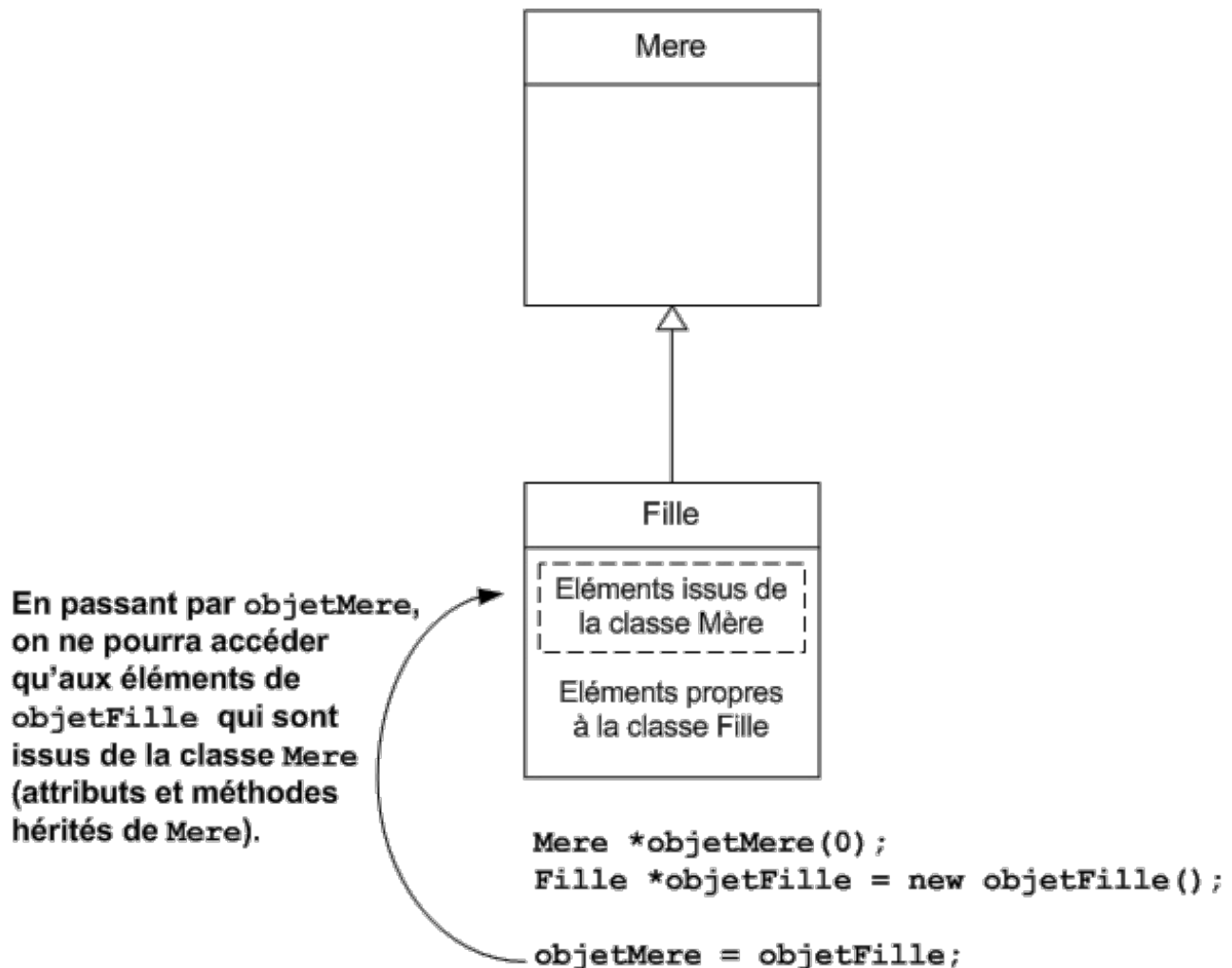
Eh ben non, moi je comprends PAS ! Je ne vois pas pourquoi ça marche si on fait :
`objetMere = objetFille;`



Là on affecte la fille à la mère, or la fille possède des attributs que la mère n'a pas. Ça devrait coincer ! L'inverse ne serait pas plus logique ?

Je vous rassure, personnellement j'ai mis des mois avant d'arriver à comprendre ce qui se passait vraiment (comment ça ça vous rassure pas ? 🤔)

Votre erreur est de croire qu'on affecte la fille à la mère. Non on n'affecte pas la fille à la mère, on substitue un pointeur (ou une référence). Déjà c'est pas du tout pareil. Les objets restent comme ils sont dans la mémoire. On fait juste pointer le pointeur sur la partie de la fille qui a été héritée. La classe fille est constituée de deux morceaux, les attributs et méthodes héritées de la mère d'une part et les attributs et méthodes propres à la classe fille. En faisant `objetMere = objetFille;`, on fait pointer le pointeur `objetMere` sur les attributs et méthodes héritées uniquement.



Voilà je peux difficilement pousser l'explication plus loin, j'espère que vous allez comprendre, sinon pas de panique j'ai survécu plusieurs mois de programmation en C++ sans bien comprendre ce qui se passait et j'en suis pas mort 🤖 (mais c'est mieux si vous comprenez c'est clair !)

En tout cas sachez que c'est une technique très utilisée, on s'en sert vraiment souvent en C++ ! Vous découvrirez bien ça avec la pratique en utilisant Qt dans la prochaine partie.

Héritage et constructeurs

Vous avez peut-être remarqué que je n'ai pas encore parlé des constructeurs dans les classes filles (Guerrier, Magicien...). C'est le moment justement de s'y intéresser 😊

On sait que `Personnage` a un constructeur (un constructeur par défaut) défini comme ceci dans le `.h` :

Code : C++

```
Personnage () ;
```

... et son implémentation dans le .cpp :

Code : C++

```
Personnage::Personnage() : m_vie(100), m_nom("Jack")
{
}
}
```

Comme vous le savez, lorsqu'on crée un objet de type Personnage, le constructeur est appelé avant toute chose.

Mais maintenant, que se passe-t-il lorsqu'on crée par exemple un Magicien qui hérite de Personnage ? Le Magicien a le droit d'avoir un constructeur lui aussi ! Est-ce que ça ne va pas interférer avec le constructeur de Personnage ? Il faut pourtant appeler le constructeur de Personnage si on veut que la vie et le nom soient initialisés !

En fait, les choses se dérouleront dans l'ordre suivant :

1. Vous demandez à créer un objet de type Magicien
2. Le compilateur appelle d'abord le constructeur de la classe mère (Personnage)
3. Puis, le compilateur appelle le constructeur de la classe fille (Magicien)

En clair, c'est d'abord le constructeur du "parent" qui est appelé, puis celui du fils, et éventuellement du petit fils (s'il y a un héritage d'héritage, comme c'est le cas avec MagicienBlanc).

Appeler le constructeur de la classe mère

Pour appeler le constructeur de Personnage en premier, il faut y faire appel depuis le constructeur de Magicien. C'est dans un cas comme ça qu'il est ~~bon~~ indispensable de se servir de la liste d'initialisation (vous savez, tout ce qui suit le symbole deux-points dans l'implémentation).

Code : C++

```
Magicien::Magicien() : Personnage(), m_mana(100)
{
}
}
```

Le premier élément de la liste d'initialisation dit de faire d'abord appel au constructeur de la classe parente Personnage. Puis, les initialisations propres au Magicien sont faites (comme l'initialisation de la mana à 100).



Lorsqu'on crée un objet de type Magicien, le compilateur appelle le constructeur par défaut de la classe mère (celui qui ne prend pas de paramètre).

Transmission de paramètres

Le gros avantage de cette technique est que l'on peut "transmettre" les paramètres du constructeur de Magicien au constructeur de Personnage. Par exemple, si le constructeur de Personnage prenait un nom en paramètre, il faudrait que le Magicien accepte lui aussi ce paramètre et le fasse passer au constructeur de Personnage :

Code : C++

```
Magicien::Magicien(string nom) : Personnage(nom), m_mana(100)
{
}
```

Bien entendu, si on veut que ça marche il faudra aussi surcharger le constructeur de Personnage pour qu'il accepte un paramètre string !

Code : C++

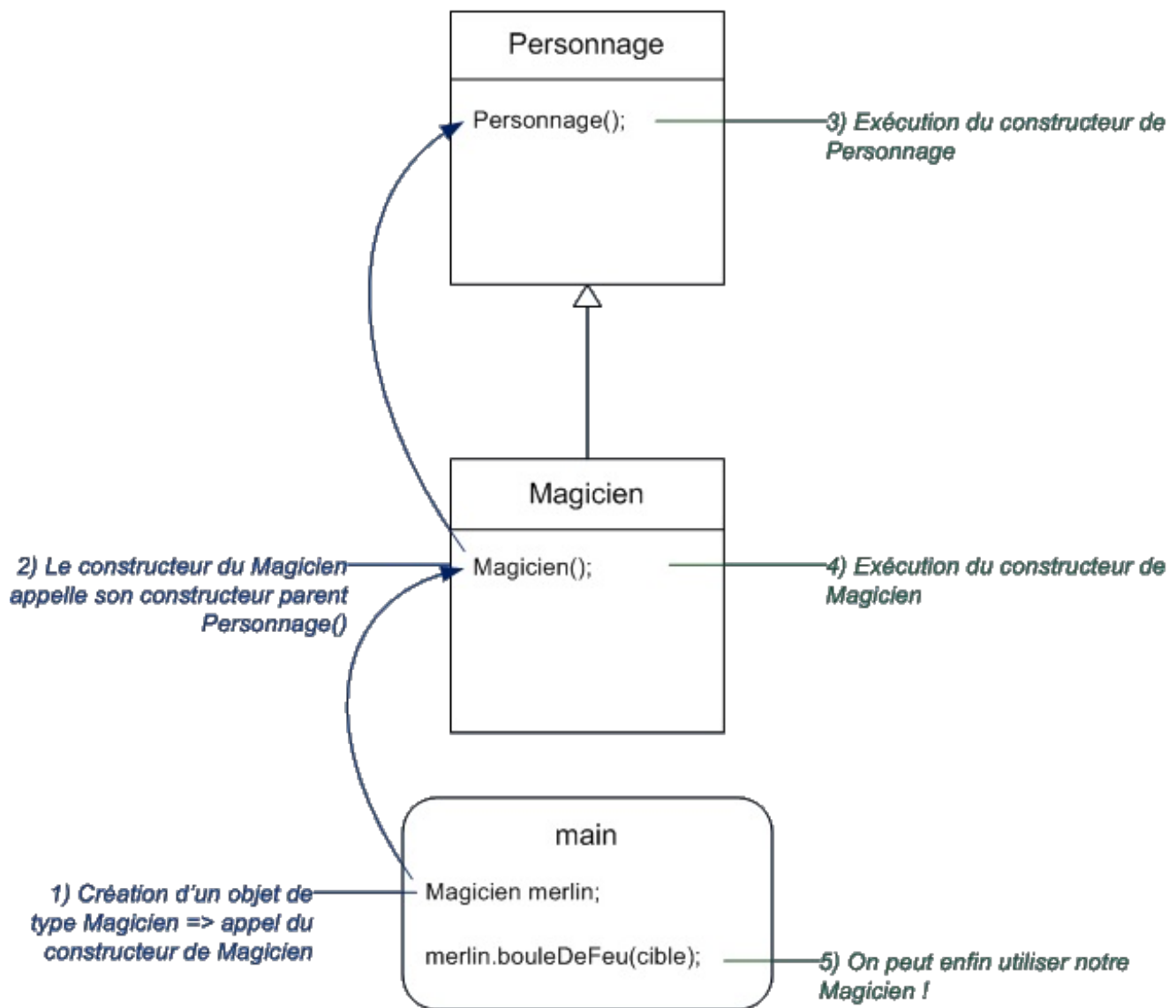
```
Personnage::Personnage(string nom) : m_vie(100), m_nom(nom)
{
}
```

Et voilà comment on fait "remonter" des paramètres d'un constructeur à un autre pour s'assurer que l'objet se crée correctement



Schéma résumé

Pour bien mémoriser ce qui se passe, rien de tel qu'un schéma résumé n'est-ce pas ? 😊



Il faut bien entendu le lire dans l'ordre pour en comprendre le fonctionnement. On commence par demander à créer un *Magicien*. "Oh mais c'est un objet" se dit le compilateur, "il faut que j'appelle son constructeur". Or, le constructeur du *Magicien* indique qu'il faut d'abord appeler le constructeur de la classe parente *Personnage*. Le compilateur va donc voir la classe parente, puis exécute son code. Il retourne ensuite au constructeur du *Magicien* et exécute son code.

Une fois que tout cela est fait, notre objet *merlin* devient utilisable et on peut enfin faire subir les pires sévices à notre cible 🐱

La portée **protected**

Il me serait vraiment impossible de vous parler d'héritage sans vous parler de la portée **protected**.

Rappel : les portées (ou droits d'accès) que vous connaissez déjà sont :



- **public** : les éléments qui suivent seront accessibles depuis l'extérieur de la classe.
- **private** : les éléments qui suivent ne seront pas accessibles depuis l'extérieur de la classe.

Je vous ai en particulier donné la règle fondamentale du C++, l'encapsulation, qui veut que l'on empêche systématiquement au monde extérieur d'accéder aux attributs de nos classes.

La portée **protected** est un autre type de droit d'accès que je classerais entre **public** (le plus permissif) et **private** (le plus restrictif). Il n'a de sens que pour les classes qui se font hériter (les classes mères) mais on peut les utiliser sur toutes les classes même quand il n'y a pas d'héritage.

Sa signification est la suivante :

protected : les éléments qui suivent ne seront pas accessibles depuis l'extérieur de la classe, *sauf* si c'est une classe fille.

Cela veut dire par exemple que si l'on met des éléments en **protected** dans la classe `Personnage`, on y aura accès dans les classes filles `Guerrier` et `Magicien`. Avec la portée **private**, on n'aurait pas pu y accéder !



En pratique, personnellement je donne toujours la portée **protected** aux attributs de mes classes. C'est comme **private** (donc ça respecte l'encapsulation) sauf que comme ça, au cas où j'hérite un jour de cette classe, j'aurai aussi directement accès aux attributs.

Cela est souvent nécessaire voire indispensable sinon on doit utiliser des tonnes d'accesseurs (méthodes `getVie()`, `getMana()`, ...) et ça rend le code bien plus lourd.

Code : C++

```
class Personnage
{
    public:
        Personnage();
        Personnage(std::string nom);
        void recevoirDegats(int degats);
        void coupDePoing(Personnage &cible) const;

    protected: // Privé, mais accessible aux éléments enfants
        (Guerrier, Magicien, ...)
        int m_vie;
        std::string m_nom;
};
```

On peut alors directement manipuler la vie et le nom dans tous les éléments enfants de `Personnage`, comme `Guerrier` et `Magicien` !

Le masquage

Terminons ce chapitre avec une notion qui nous servira dans la suite : le masquage.

Une fonction de la classe mère

Il serait intéressant pour notre petit RPG que nos personnages aient le moyen de se présenter. Comme c'est une action que devraient pouvoir réaliser tous les personnages quels que soient leurs rôles militaires, la fonction `sePresenter()` va dans la classe `Personnage`.

Code : C++

```
class Personnage
{
    public:
        Personnage();
        Personnage(std::string nom);
        void recevoirDegats(int degats);
        void coupDePoing(Personnage& cible) const;

        void sePresenter() const;

    protected:
        int m_vie;
        std::string m_nom;
};
```



Remarquez le **const** qui indique que le personnage ne sera pas modifié quand il se présentera. Vous en avez maintenant l'habitude, mais j'aime bien vous rafraîchir la mémoire. 😊

Et dans le fichier `.cpp` :

Code : C++

```
void Personnage::sePresenter() const
{
    cout << "Bonjour, je m'appelle " << m_nom << "." << endl;
    cout << "J'ai encore " << m_vie << " points de vie." << endl;
}
```

On peut donc écrire un *main* du type :

Code : C++

```
int main()
{
    Personnage marcel("Marcel");
    marcel.sePresenter();

    return 0;
}
```

Ce qui nous donne évidemment le résultat suivant :

Code : Console

```
Bonjour, je m'appelle Marcel.
J'ai encore 100 points de vie.
```

La fonction est héritée dans les classes filles

Vous le savez déjà, un Guerrier EST UN Personnage et par conséquent, il peut également se présenter.

Code : C++

```
int main() {
    Guerrier lancelet("Lancelot du Lac");
    lancelet.sePresenter();

    return 0;
}
```

Donnera :

Code : Console

```
Bonjour, je m'appelle Lancelot du Lac.
J'ai encore 100 points de vie.
```

Jusque là, rien de bien particulier et de difficile. 😊

Le masquage

Imaginons maintenant que les guerriers aient une manière différente de se présenter. Ils doivent en plus dire qu'ils sont guerriers. Nous allons donc écrire une version différente de la fonction `sePresenter()` spécialement pour eux :

Code : C++

```
void Guerrier::sePresenter() const
{
    cout << "Bonjour, je m'appelle " << m_nom << "." << endl;
    cout << "J'ai encore " << m_vie << " points de vie." << endl;
    cout << "Je suis un Guerrier redoutable." << endl;
}
```



Mais, il y aura deux fonctions avec le même nom et les mêmes arguments dans la classe ! C'est interdit !

Vous avez tort et raison. Deux fonctions ne peuvent avoir la même signature (nom et type des arguments). Mais dans le cadre des classes c'est différent. La fonction de la classe `Guerrier` va remplacer celle héritée de la classe `Personnage`.

Si l'on exécute le même `main()` qu'avant, on obtient cette fois le résultat souhaité.

Code : Console

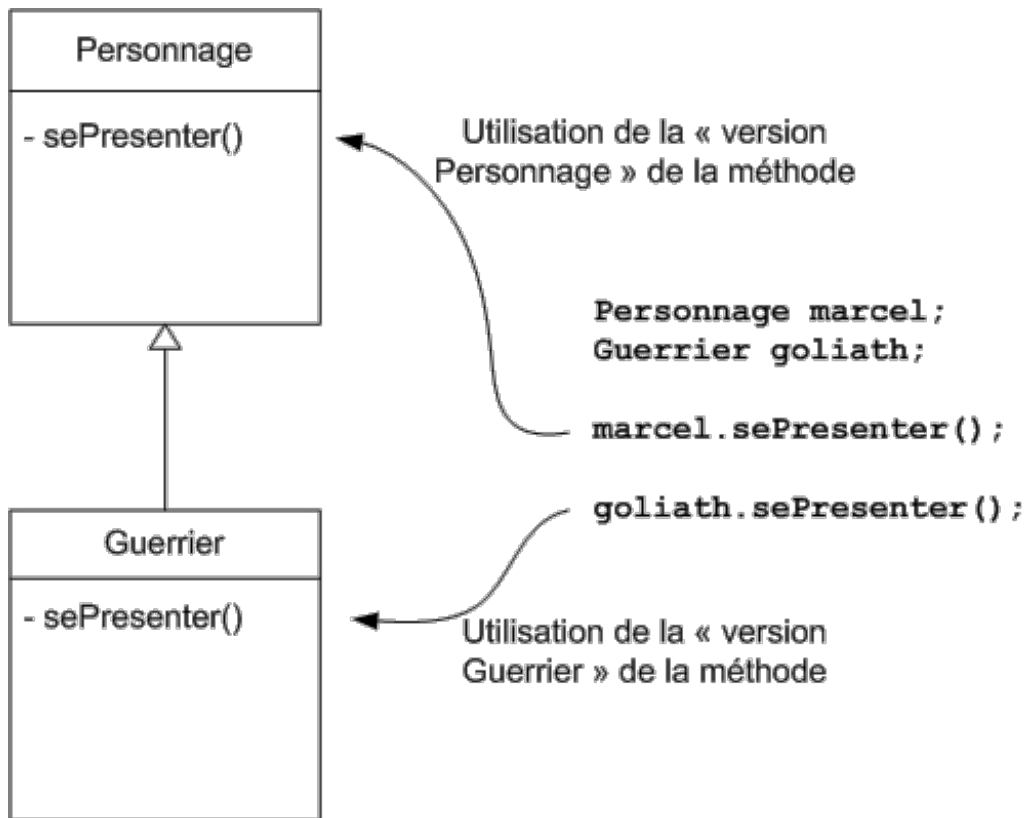
```
Bonjour, je m'appelle Lancelot du Lac.
J'ai encore 100 points de vie.
Je suis un guerrier redoutable.
```

Quand on écrit une fonction qui a le même nom que celle héritée de la classe mère, on parle de **masquage**. La fonction héritée de `Personnage` est masquée ; elle est cachée.



Pour masquer une fonction, il suffit qu'elle ait le même nom qu'une autre fonction héritée. Le nombre et le type des arguments ne joue aucun rôle.

C'est bien pratique ça ! Quand on fait un héritage, la classe fille reçoit automatiquement toutes les méthodes de la classe mère. Si une de ces méthodes ne nous plait pas, on la réécrit dans la classe fille. Le compilateur saura quelle version appeler. Si c'est un `Guerrier`, il utilise la "version `Guerrier`" de `sePresenter()` et si c'est un `Personnage` ou un `Magicien`, il utilise la version de base.



Gardez bien ce schéma en mémoire, il nous sera utile dans le prochain chapitre. 😊

Économiser du code

Ce qu'on a écrit est bien, mais on peut faire encore mieux. Si l'on regarde, la fonction `sePresenter()` de la classe `Guerrier` a deux lignes identiques à ce qu'il y a dans la même fonction de la classe `Personnage`. On pourrait donc économiser des lignes de code en appelant la fonction masquée.



Économiser des lignes de code est souvent une bonne attitude à avoir. Le code est ainsi plus facilement maintenable. Et souvenez-vous être fainéant est une qualité importante des programmeurs. 😎

On aimerait donc écrire quelque chose du genre :

Code : C++

```

void Guerrier::sePresenter() const
{
    appel_a_la_fonction_masquee(); //Cela afficherait les
    informations de base
    cout << "Je suis un Guerrier redoutable." << endl; //Et ensuite
    les informations spécifiques
}
  
```

Il faudrait donc un moyen d'appeler la fonction de la classe mère.

Le démasquage

On aimerait appeler la fonction dont le nom complet est : `Personnage::sePresenter()`.

Essayons donc.

Code : C++

```
void Guerrier::sePresenter() const
{
    Personnage::sePresenter();
    cout << "Je suis un Guerrier redoutable." << endl;
}
```

Et c'est magique, cela donne exactement ce que l'on espérait. 🧙

Code : Console

```
Bonjour, je m'appelle Lancelot du Lac.
J'ai encore 100 points de vie.
Je suis un guerrier redoutable.
```

On parle dans ce cas de **démasquage** puisqu'on a pu utiliser une fonction qui était masquée.

On a utilisé ici l'**opérateur ::** appelé **opérateur de résolution de portée**. Il sert à déterminer quelle fonction (ou variable) utiliser quand il y a ambiguïté ou si il y a plusieurs possibilités.

Ce chapitre en impose peut-être un peu par sa taille, mais ne vous y fiez pas ce sont surtout les schémas qui prennent de la place.



D'ailleurs, j'ai volontairement évité de trop montrer de codes sources complets différents et j'ai préféré que vous vous focalisiez sur ces schémas. C'est ce qu'on retient le mieux en général, et ça permet de bien se repérer. La pratique viendra dans la partie sur la librairie Qt.

Ceci étant, peut-être que vous aimeriez avoir le code source complet de mes exemples (Personnage, Guerrier, Magicien...). Ce code n'est pas complet, certaines méthodes ne sont pas écrites, il ne fait rien d'extraordinaire. Mais il compile, et ça vous permettra peut-être de finir de mettre de l'ordre dans vos idées.

Voici donc le code source :

[Télécharger le code source complet \(3 Ko\)](#)

Bon bidouillage ! 😊

Le polymorphisme

Vous avez bien compris le chapitre sur l'héritage ? C'était un chapitre relativement difficile. Je ne veux pas vous faire peur, mais celui que vous êtes en train de lire est du même acabit. C'est sans doute le chapitre le plus complexe de tout le cours, mais vous allez voir qu'il va nous ouvrir de nouveaux horizons très intéressants.

Mais au fait, de quoi allons-nous parler ? Le titre est simplement "le polymorphisme", ce qui ne nous avance pas vraiment. 😞 Si vous avez fait un peu de grec, vous êtes peut-être à même de décortiquer un petit peu ce mot. « *Poly* » signifie « *plusieurs* » comme dans polygone ou polytechnique et « *morphe* » signifie « *forme* » comme... euh... amorphe ou zoomorphe. 😊

Nous allons donc parler de choses ayant plusieurs formes. Ou pour utiliser des termes d'informatique, nous allons créer du code fonctionnant de manière différente selon le type qui l'utilise.

Nous verrons plus tard dans ce cours un autre moyen de faire cela, grâce aux *templates*. Pour l'instant, nous allons nous contenter de créer des fonctions qui s'exécutent différemment selon qu'on utilise un objet d'une classe mère ou d'une classe fille.

Courage ! C'est le dernier chapitre vraiment difficile avant d'attaquer la pratique du C++ et notamment la création de programmes avec des vraies fenêtres à la place de cette vieille console.



Je vous conseille vivement de relire le chapitre sur les pointeurs avant de continuer.

La résolution des liens

Commençons en douceur avec un peu d'héritage tout simple. 😊 Vous en avez marre de notre RPG ? Moi aussi. Prenons un autre exemple pour varier un peu. Attaquons donc la création d'un programme de gestion d'un garage et des véhicules qui y sont stationnés. Imaginons que notre fier garagiste sache réparer à la fois des voitures et des motos. Dans son programme, il aurait les classes suivantes : Vehicule, Voiture et Moto.

Code : C++ - Définition des classes

```
class Vehicule
{
    public:
        void affiche() const;    //Affiche une description du Vehicule

    protected:
        int m_prix;            //Chaque véhicule a un prix
};

class Voiture : public Vehicule //Une Voiture EST UN Vehicule
{
    public:
        void affiche() const;

    private:
        int m_portes;          //Le nombre de portes de la voiture
};

class Moto : public Vehicule //Une Moto EST UN Vehicule
{
    public:
        void affiche() const;

    private:
        double m_vitesse;      //La vitesse maximale de la moto
};
```

L'exemple est simplifié au maximum. Il manque bien sûr beaucoup de méthodes, d'attributs et les constructeurs. Je vous laisse compléter selon vos envies.

Le corps des fonctions `affiche()` est le suivant :

Code : C++ - Corps des fonctions `affiche()`

```
void Vehicule::affiche() const
{
    cout << "Ceci est un vehicule." << endl;
}

void Voiture::affiche() const
{
    cout << "Ceci est une voiture." << endl;
}

void Moto::affiche() const
{
    cout << "Ceci est une moto." << endl;
}
```

Chaque classe affiche donc un message différent. Et si vous avez bien suivi le chapitre précédent, vous aurez reconnu que j'utilise ici le masquage pour redéfinir la fonction `affiche()` de `Vehicule` dans les deux classes filles.


Essayons donc ces fonctions avec un petit main tout bête.

Code : C++

```
int main()
{
    Vehicule v;
    v.affiche();    //Affiche "Ceci est un vehicule. "

    Moto m;
    m.affiche();    //Affiche "Ceci est une moto. "

    return 0;
}
```

Je vous invite à tester, vous ne devriez rien observer de particulier. Mais ça va venir. 

La résolution statique des liens

Créons une fonction supplémentaire qui reçoit en paramètre un `Vehicule` et modifions le main de sorte à utiliser la fonction :

Code : C++ - Une fonction en apparence inoffensive

```
void presenter(Vehicule v)    //Présente le véhicule passé en
                               argument
{
    v.affiche();
}

int main()
{
    Vehicule v;
    presenter(v);

    Moto m;
    presenter(m);

    return 0;
}
```

Et testons là. A priori, rien n'a changé. Les messages affichés devraient être les mêmes. Voyons ça :

Code : Console

```
Ceci est un vehicule.  
Ceci est un vehicule.
```



Le message n'est pas correct pour la moto ! C'est comme si lors du passage dans la fonction la vraie nature de la moto s'était perdue et qu'elle était redevenue un simple véhicule.



Comment est-ce possible ?

Comme il y a une relation d'héritage, nous savons qu'une moto EST UN véhicule, un véhicule amélioré en quelque sorte puisqu'il possède un attribut supplémentaire. La fonction `présenter()` reçoit en argument un `Vehicule`. Ça peut être un vrai objet de type `Vehicule`, mais aussi une `Voiture` ou comme dans l'exemple, une `Moto`. Souvenez-vous de la dérivation de type introduite au chapitre précédent.

Ce qui est important, c'est que pour le compilateur, à l'intérieur de la fonction, il manipule un `Vehicule`. Peu importe sa vraie nature. Il va donc appeler la "version `Vehicule`" de la méthode `afficher()` et pas la "version `Moto`" comme on aurait pu l'espérer.

Dans l'exemple du chapitre précédent, c'est la bonne version qui était appelée puisque à l'intérieur de la fonction, le compilateur savait si il avait affaire à un simple personnage ou à un guerrier. Ici, dans la fonction `présenter()`, pas moyen de savoir ce que sont réellement les véhicules reçus en argument.

En termes techniques, on parle de **résolution statique des liens**. La fonction reçoit un `Vehicule`, c'est donc toujours la "version `Vehicule`" des méthodes qui sera utilisée.

C'est le type de la variable qui détermine quelle fonction membre appeler et pas sa vraie nature.

Mais vous vous doutez bien que si je vous parle de tout ça, c'est qu'il y a moyen de changer ce comportement. 🤔

La résolution dynamique des liens

Ce qu'on aimerait nous, c'est que la fonction `présenter()` appelle la bonne version de la méthode. C'est-à-dire qu'il faut que la fonction connaisse la vraie nature du `Vehicule`. C'est ce qu'on appelle la **résolution dynamique des liens**. Lors de l'exécution, le programme va utiliser la bonne version des méthodes car il saura si l'objet est de type mère ou de type fille.

Pour faire cela, il faut deux *ingrédients* :

- Utiliser un pointeur ou une référence.
- Utiliser des méthodes virtuelles.



Si ces deux ingrédients ne sont pas réunis, alors on retombe dans le premier cas et l'ordinateur n'aura aucun moyen d'appeler la bonne méthode.

Les fonctions virtuelles

Je vous ai donné la liste des ingrédients, allons-y pour la préparation du menu. Commençons par les méthodes virtuelles.

Déclarer une méthode virtuelle...

Ça a l'air effrayant en le lisant, mais c'est très simple. Il suffit d'ajouter le mot-clé **virtual** dans le prototype de la classe (dans le fichier .h donc). Donc pour notre garage, cela donne :

Code : C++ - Quelques méthodes virtuelles

```
class Vehicule
```

```

{
    public:
        virtual void affiche() const;    //Affiche une description du
        Vehicule

    protected:
        int m_prix;    //Chaque véhicule a un prix
};

class Voiture: public Vehicule //Une Voiture EST UN Vehicule
{
    public:
        virtual void affiche() const;

    private:
        int m_portes;    //Le nombre de portes de la voiture
};

class Moto : public Vehicule //Une Moto EST UN Vehicule
{
    public:
        virtual void affiche() const;

    private:
        double m_vitesse;    //La vitesse maximale de la moto
};

```



Il n'est pas nécessaire de mettre le **virtual** devant les méthodes des classes filles. Elles sont automatiquement virtuelles par héritage. Personnellement, je préfère le mettre pour me souvenir de leur particularité.

Jusque-là rien de bien difficile. Notez bien qu'il n'est pas nécessaire que toutes les méthodes soient virtuelles. Une classe peut très bien proposer des fonctions "normales" et d'autres virtuelles.



Il ne faut pas mettre **virtual** dans le fichier .cpp, mais uniquement dans le .h. Si vous essayez, votre compilateur se vengera en vous insultant copieusement !

... et utiliser une référence

Le deuxième ingrédient est un pointeur ou une référence. Vous êtes certainement comme moi, vous préférez la simplicité et par conséquent les références. On ne va quand même pas s'embêter avec des pointeurs juste pour le plaisir. 😊

Réécrivons donc la fonction `presenter()` avec comme argument une référence.

Code : C++

```

void presenter(Vehicule const& v)    //Présente le véhicule passé en
argument
{
    v.affiche();
}

int main()    //Rien n'a changé dans le main()
{
    Vehicule v;
    presenter(v);

    Moto m;
    presenter(m);

    return 0;
}

```

}



J'ai aussi ajouté un **const**. Comme on ne modifie pas l'objet dans la fonction, autant le faire savoir au compilateur et au programmeur en déclarant la référence constante.

Voilà. Il ne nous reste plus qu'à tester.

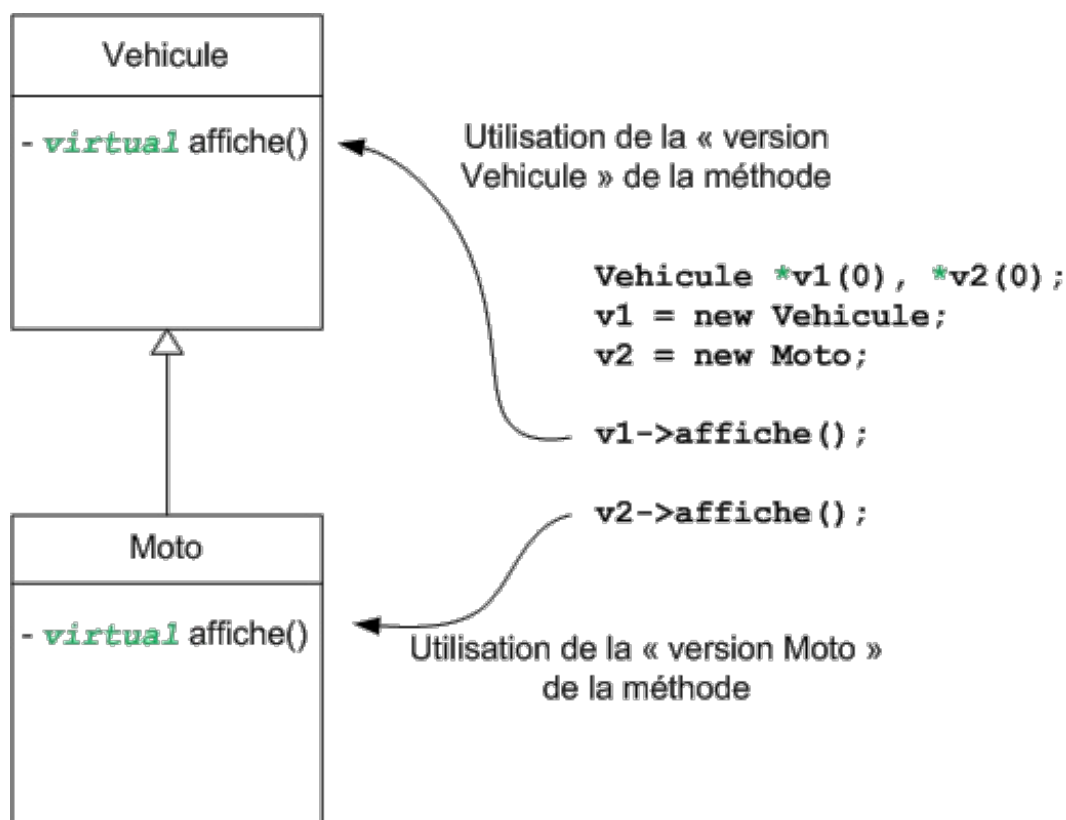
Code : Console

```
Ceci est un vehicule.  
Ceci est une moto.
```



Ça marche ! La fonction `presenter()` a bien appelé la bonne version de la méthode. En utilisant des fonctions virtuelles et une référence sur l'objet, la fonction `presenter()` a pu correctement choisir la méthode à appeler.

On aurait obtenu le même comportement avec des pointeurs à la place des références. Comme sur le schéma suivant.



Un même bout de code a eu deux comportements différents selon le type passé en argument. C'est donc du polymorphisme. On dit aussi que les méthodes `affiche()` ont un **comportement polymorphique**.

Les méthodes spéciales

Bon assez parlé. A mon tour de vous poser une petite question de théorie :



Quelles sont les méthodes d'une classe qui ne sont jamais héritées ?

Secret (cliquez pour afficher)

- Tous les constructeurs.

- Le destructeur.

Vous avez trouvé ? C'est bien. Toutes les autres méthodes peuvent être héritées et peuvent avoir un comportement polymorphique si on le souhaite. Qu'en est-il pour ces méthodes spéciales ?

Le cas des constructeurs

Un constructeur virtuel a-t-il du sens ? Non ! Quand je veux construire un véhicule quelconque, je sais lequel je veux construire. Je peux donc à la compilation déjà savoir quel véhicule construire. Je n'ai pas besoin de résolution dynamique des liens et par conséquent pas besoin de virtualité.

Un constructeur ne peut pas être virtuel.

Et cela va même plus loin. Quand je suis dans le constructeur, je sais quel type je construis, je n'ai donc à nouveau pas besoin de résolution dynamique des liens. D'où la règle suivante :

On ne peut pas appeler de méthodes virtuelles dans un constructeur. Si on essaye quand même, la résolution dynamique des liens ne se fait pas.

Le cas du destructeur

Ici, c'est un petit peu plus compliqué. Malheureusement. 🤔

Créons un petit programme utilisant nos véhicules et des pointeurs puisque c'est un des ingrédients du polymorphisme.

Code : C++

```
int main()
{
    Vehicule *v(0);
    v = new Voiture;    //On crée une Voiture et on met son adresse
                        //dans un pointeur de Vehicule

    v->affiche();       //On affiche "Ceci est une voiture."

    delete v;          //Et on détruit notre voiture

    return 0;
}
```

Nous avons un pointeur et une méthode virtuelle. La ligne `v->affiche()` va donc afficher le message que l'on souhaitait. Le problème de ce programme se situe au moment du **delete**. Nous avons un pointeur, mais la méthode appelée n'est pas virtuelle. C'est donc le destructeur de `Vehicule` qui est appelé et pas celui de `Voiture` !

Dans ce cas, cela ne porte pas vraiment à conséquence. Le programme ne va pas planter. Mais imaginez que vous deviez écrire une classe pour le maniement des moteurs électriques d'un robot. Si c'est le mauvais destructeur qui est appelé, peut-être que vos moteurs ne vont pas s'arrêter. Cela peut vite devenir dramatique.

Il faut donc impérativement appeler le bon destructeur. Et pour se faire, une seule solution : rendre le destructeur virtuel ! Ce qui nous permet de formuler une nouvelle règle importante :

Un destructeur doit toujours être virtuel si on utilise le polymorphisme.

Le code amélioré

Ajoutons donc des constructeurs et des destructeurs à nos classes. Tout sera alors correct.

Code : C++

```

class Vehicule
{
    public:
        Vehicule(int prix);           //Construit un véhicule d'un
        certain prix
        virtual void affiche() const;
        virtual ~Vehicule();         //Remarquez le 'virtual' ici

    protected:
        int m_prix;
};

class Voiture: public Vehicule
{
    public:
        Voiture(int prix, int portes); //Construit une voiture dont on
        fournit le prix et le nombre de portes
        virtual void affiche() const;
        virtual ~Voiture();

    private:
        int m_portes;
};

class Moto : public Vehicule
{
    public:
        Moto(int prix, double vitesseMax); //Construit une moto d'un
        prix donné et ayant une certaine vitesse maximale
        virtual void affiche() const;
        virtual ~Moto();

    private:
        double m_vitesse;
};

```

Il faut bien sûr également compléter le fichier source :

Code : C++

```

Vehicule::Vehicule(int prix)
    :m_prix(prix)
{}

void Vehicule::affiche() const //J'en profite pour modifier un peu
les fonctions d'affichage
{
    cout << "Ceci est un vehicule coutant " << m_prix << " euros."
    << endl;
}

Vehicule::~Vehicule() //Même si le destructeur ne fait rien, on
doit le mettre !
{}

Voiture::Voiture(int prix, int portes)
    :Vehicule(prix), m_portes(portes)
{}

void Voiture::affiche() const
{
    cout << "Ceci est une voiture avec " << m_portes << " portes et
    coutant " << m_prix << " euros." << endl;
}

```

```

Voiture::~Voiture()
{

}

Moto::Moto(int prix, double vitesseMax)
    :Vehicule(prix), m_vitesse(vitesseMax)
{

}

void Moto::affiche() const
{
    cout << "Ceci est une moto allant a " << m_vitesse << " km/h et
coutant " << m_prix << " euros." << endl;
}

Moto::~Moto()
{
}

```

Nous sommes donc prêt à aborder un exemple concret d'utilisation du polymorphisme. Attachez vos ceintures ! 😊

Les collections hétérogènes

Je vous ai dit tout au début du chapitre que nous voulions créer un programme de gestion d'un garage. Nous allons donc devoir gérer une collection de voitures et de motos. Nous allons donc utiliser des ... tableaux dynamiques !

Code : C++ - Des listes de voitures et de motos

```

vector<Voiture> listeVoitures;
vector<Moto> listeMotos;

```

Bien ! Mais pas optimal. 😞 Si notre ami garagiste commence à recevoir des commandes pour des scooters, des camions, des fourgons, des vélos, etc. il va falloir déclarer beaucoup de vectors. Cela veut dire qu'il va falloir faire de grosses modifications au code à chaque fois qu'un nouveau type de véhicule apparaît.

Le retour des pointeurs

Il serait bien mieux de mettre le tout dans un seul tableau ! Comme les motos et les voitures sont des véhicules, on peut déclarer un tableau de véhicules et mettre des motos dedans.

Mais si on fait ça, alors nous allons perdre la vraie nature des objets. Souvenez-vous des deux ingrédients du polymorphisme ! Il nous faut donc un tableau de pointeurs ou un tableau de références. On ne peut pas créer un tableau de références ([Rappelez-vous](#), les références ne sont que des étiquettes), nous allons donc devoir utiliser des pointeurs. 😞

Vous vous rappelez du chapitre sur les pointeurs ? Je vous avais présenté trois cas d'utilisation. En voici donc un quatrième. J'espère que vous ne m'en voulez pas trop de ne pas en avoir parlé avant... 😊

Code : C++ - Un tableau de pointeurs sur des véhicules

```

int main()
{
    vector<Vehicule*> listeVehicules;
    return 0;
}

```

C'est ce qu'on appelle une **collection hétérogène** puisqu'elle contient d'une certaine manière des types différents.

Utiliser la collection

Commençons par remplir notre tableau. Comme nous allons accéder à nos véhicules uniquement via les pointeurs, nous n'avons pas besoin d'étiquettes sur nos objets, nous pouvons utiliser l'allocation dynamique pour créer nos objets. En plus, cela nous permet d'avoir directement un pointeur à mettre dans notre vector.

Code : C++

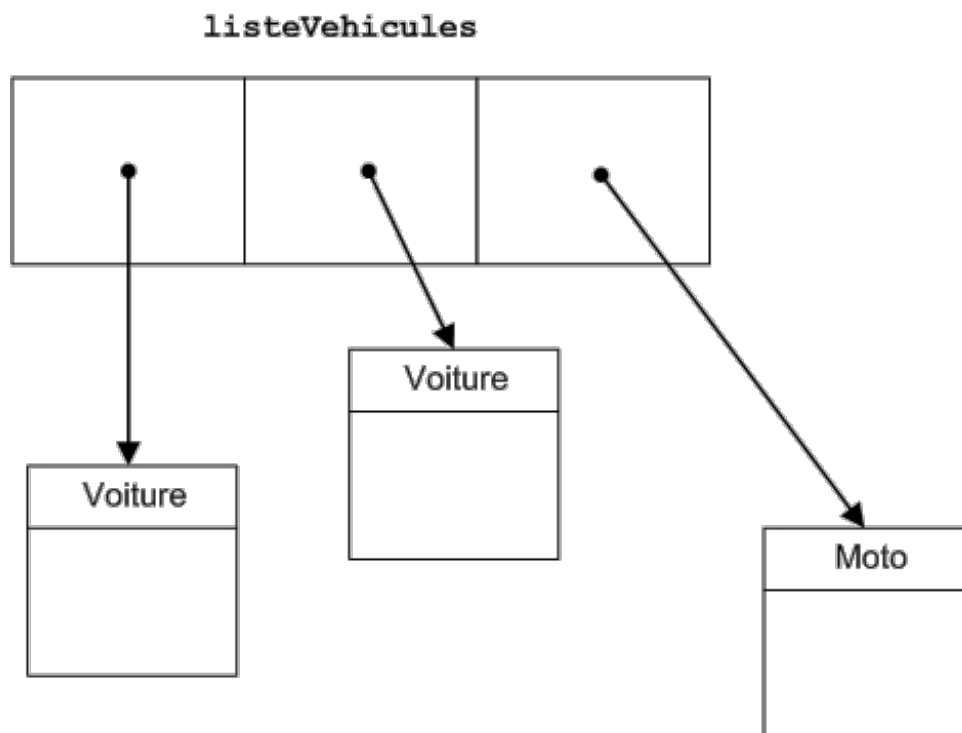
```
int main()
{
    vector<Vehicule*> listeVehicules;

    listeVehicules.push_back(new Voiture(15000, 5)); //J'ajoute une
    voiture valant 15000 euros                          // et ayant 5
    portes à ma collection de véhicules
    listeVehicules.push_back(new Voiture(12000, 3)); //...
    listeVehicules.push_back(new Moto(2000, 212.5)); //Une moto à
    2000 euros allant à 212.5 km/h

    //On utilise les voitures et les motos

    return 0;
}
```

Voyons à quoi ressemble notre tableau :



Les voitures et motos ne sont pas réellement dans les cases. Ce sont des pointeurs. Mais en suivant les flèches, on arrive à accéder aux véhicules.

Bien ! Mais nous venons de faire une grosse faute ! 😞 Chaque fois que l'on utilise **new**, il faut utiliser **delete** pour vider la mémoire. Nous allons donc devoir utiliser une boucle pour libérer la mémoire allouée.

Code : C++

```
int main()
{
    vector<Vehicule*> listeVehicules;
```

```

listeVehicules.push_back(new Voiture(15000, 5));
listeVehicules.push_back(new Voiture(12000, 3));
listeVehicules.push_back(new Moto(2000, 212.5));

//On utilise les voitures et les motos

for(int i(0); i<listeVehicules.size(); ++i)
{
    delete listeVehicules[i];    //On libère la i-ème case
    //mémoire allouée
    listeVehicules[i] = 0;        //Et on met le pointeur à 0
    //pour éviter les soucis
}

return 0;
}

```

Il ne nous reste plus qu'à utiliser nos objets. Comme c'est un exemple basique, ils ne savent faire qu'une seule chose: afficher des informations. Mais essayons quand même !

Code : C++

```

int main()
{
    vector<Vehicule*> listeVehicules;

    listeVehicules.push_back(new Voiture(15000, 5));
    listeVehicules.push_back(new Voiture(12000, 3));
    listeVehicules.push_back(new Moto(2000, 212.5));

    listeVehicules[0]->affiche();    //On affiche les informations
    //de la première voiture
    listeVehicules[2]->affiche();    //et celles de la moto

    for(int i(0); i<listeVehicules.size(); ++i)
    {
        delete listeVehicules[i];    //On libère la i-ème case
        //mémoire allouée
        listeVehicules[i] = 0;        //Et on met le pointeur à 0
        //pour éviter les soucis
    }

    return 0;
}

```

Je vous invite, comme toujours, à tester. Voici ce que vous devriez obtenir :

Code : Console

```

Ceci est une voiture avec 5 portes valant 15000 euros.
Ceci est une moto allant a 212.5 km/h et valant 2000 euros.

```

Ce sont les bonnes versions des méthodes qui sont appelées ! 😊 Ce ne devrait pas être une surprise à ce stade. Nous avons des pointeurs (ingrédient 1) et des méthodes virtuelles (ingrédient 2).

Je vous propose d'améliorer un peu ce code en ajoutant les éléments suivants :

- Une classe `Camion` qui aura comme attribut le poids qu'il peut transporter.
- Un attribut représentant l'année de fabrication du véhicule. Ajoutez aussi des méthodes pour afficher cette information.

- Une classe Garage qui aura comme attribut le `vector<Vehicule*>` et proposerait des méthodes pour ajouter/supprimer des véhicules ou pour afficher des informations sur tous les éléments contenus.
- Une méthode `nbrRoues()` qui renvoie le nombre de roues des différents véhicules.

Après ce léger entraînement, terminons ce chapitre avec une évolution de notre petit programme.

Les fonctions virtuelles pures

Avez-vous essayé de programmer la méthode `nbrRoues()` du mini-exercice ? Non ! Il est encore temps de le faire. Elle va beaucoup nous intéresser dans la suite. 😊

Le problème des roues

Comme c'est un peu répétitif, je vous donne ma version de la fonction pour les classes véhicules et voiture uniquement.

Code : C++

```
class Vehicule
{
public:
    Vehicule(int prix);
    virtual void affiche() const;
    virtual int nbrRoues() const; //Affiche le nombre de roues du
    véhicule
    virtual ~Vehicule();

protected:
    int m_prix;
};

class Voiture : public Vehicule
{
public:
    Voiture(int prix, int portes);
    virtual void affiche() const;
    virtual int nbrRoues() const; //Affiche le nombre de roues de la
    voiture
    virtual ~Voiture();

private:
    int m_portes;
};
```

Du côté du `.h`, pas de soucis. C'est le corps des fonctions qui risque de poser problème.

Code : C++

```
int Vehicule::nbrRoues() const
{
    //Que mettre ici ???
}

int Voiture::nbrRoues() const
{
    return 4;
}
```

Vous l'aurez compris, on ne sait pas vraiment quoi mettre dans la "version Vehicule" de la méthode. Les voitures ont 4 roues et les motos 2, mais pour un véhicule en général, on ne peut rien dire ! On aimerait bien ne rien mettre ici ou carrément supprimer la fonction puisqu'elle n'a pas de sens.

Mais si on ne déclare pas la fonction dans la classe mère, alors on ne pourra pas l'utiliser depuis notre collection hétérogène. Il

nous faut donc la garder ou au minimum dire qu'elle existe mais qu'on n'a pas le droit de l'utiliser. On souhaiterait ainsi dire au compilateur :

« Dans toutes les classes filles de *Vehicule*, il y aura une fonction nommée *nbrRoues()* qui renvoie un *int* et qui ne prend aucun argument, mais dans la classe *Vehicule*, cette fonction n'existe pas. »

C'est ce qu'on appelle une **méthode virtuelle pure**.

Pour déclarer une telle méthode, rien de plus simple. Il suffit d'ajouter `= 0` à la fin du prototype.

Code : C++ - Une fonction virtuelle pure

```
class Vehicule
{
    public:
        Vehicule(int prix);
        virtual void affiche() const;
        virtual int nbrRoues() const = 0;           //Affiche le nombre de
        roues du véhicule
        virtual ~Vehicule();

    protected:
        int m_prix;
};
```

Et évidemment, on n'a rien à écrire dans le `.cpp` puisque justement on ne sait pas quoi y mettre. On peut carrément supprimer complètement la méthode. L'important étant que son prototype soit présent dans le `.h`.

Les classes abstraites

Une classe qui possède au moins une méthode virtuelle pure est une **classe abstraite**. Notre classe *Vehicule* est donc une classe abstraite. 😊

Pourquoi donner un nom spécial à ces classes ? Eh bien, parce qu'elles ont une règle bien particulière :

On ne peut pas créer d'objet à partir d'une classe abstraite.

😬😬 Oui, oui, vous avez bien lu ! La ligne suivante ne compilera pas.

Code : C++

```
Vehicule v(10000); //Création d'un véhicule valant 10000 euros.
```

Dans le jargon des programmeurs, on dit qu'on ne peut pas créer d'instance d'une classe abstraite.

La raison est simple. Si je peux créer un *Vehicule*, alors je pourrais essayer d'appeler la fonction *nbrRoues()* qui n'a pas de corps et ceci n'est pas possible.

Je peux par contre tout à fait écrire le code suivant :

Code : C++

```
int main()
{
    Vehicule* ptr(0);    // Un pointeur sur un véhicule

    Voiture caisse(20000,5); // On crée une voiture, ceci est
    autorisé puisque toutes les fonctions ont un corps.
```

```
ptr = &caisse;           // On fait pointer le pointeur sur la
voiture.

cout << ptr->nbrRoues() << endl; // Dans la classe fille
nbrRoues() existe donc ceci est autorisé.

return 0;
}
```

Ici, l'appel à la méthode `nbrRoues()` est polymorphique, puisque nous avons un pointeur et que notre méthode est virtuelle. C'est donc la "version `Voiture`" qui est appelée. Donc même si la "version `Vehicule`" n'existe pas, il n'y a pas de problèmes.

Si l'on veut créer une nouvelle sorte de `Vehicule` (`Camion` par exemple), on sera obligé de redéfinir la fonction `nbrRoues()`, sinon cette dernière sera virtuelle pure par héritage et par conséquent la classe abstraite aussi.

On peut résumer les fonctions virtuelles de la manière suivante :

- Une **méthode virtuelle** peut être redéfinie dans une classe fille.
- Une **méthode virtuelle pure** doit être redéfinie dans une classe fille.

Dans la bibliothèque Qt, que nous allons très bientôt aborder, il y a beaucoup de classes abstraites. Il existe par exemple une classe par sorte de bouton, c'est-à-dire une classe pour les boutons normaux, une pour les cases à cocher, etc. Toutes ces classes héritent d'une classe nommée `QAbstractButton`, qui regroupe des propriétés communes à tous les boutons (taille, texte, ...). Mais comme on ne veut pas autoriser les utilisateurs à mettre des `QAbstractButton` sur leurs fenêtres, les créateurs de la bibliothèque ont rendu cette classe abstraite. 🤔

Pffouu... 🤔

Nous voilà arrivé au bout. C'était un chapitre vraiment complexe et vous êtes certainement dans la même situation que moi lorsqu'on m'a parlé pour la première fois du polymorphisme, vous êtes perplexe et pas sûr d'avoir bien compris. Vous verrez, avec un peu de pratique, cela va presque devenir naturel.

Courage ! Nous sommes proches de la fin des chapitres théoriques. Vous verrez que faire des programmes avec des fenêtres sera un vrai régal après ça. 😊

Éléments statiques et amitié

Vous tenez le coup ? 😊

Courage, vos efforts seront bientôt largement récompensés.

Ce chapitre va d'ailleurs vous permettre de souffler un peu. Vous allez découvrir quelques notions spécifiques aux classes en C++ : les attributs et méthodes statiques et l'amitié. Ce sont ce que j'appellerais des "points particuliers" du C++. Ce ne sont pas des détails pour autant, ce sont des choses à connaître.

Car oui, tout ce que je vous apprend ici, vous allez en avoir besoin et vous allez largement le réutiliser. Je suis sûr aussi que vous en comprendrez mieux l'intérêt lorsque vous pratiquerez pour de bon.

N'allez pas croire que les programmeurs ont inventé des trucs un peu complexes comme ça juste pour le plaisir de programmer de façon tordue 😊

Les méthodes statiques

Ah les méthodes statiques... Alors ça, c'est un peu spécial 😊

Ce sont des méthodes qui appartiennent à la classe mais pas aux objets instanciés à partir de la classe... En fait, ce sont de bêtes "fonctions" rangées dans des classes qui n'ont pas accès aux attributs de la classe. Ça s'utilise d'une manière un peu particulière.

Le mieux est encore un exemple je pense !

Créer une méthode statique

Dans le .h, le prototype d'une méthode statique ressemble à ceci :

Code : C++

```
class MaClasse
{
    public:
        MaClasse();
        static void maMethode();
};
```

Son implémentation dans le .cpp ne possède pas en revanche de mot-clé *static* :

Code : C++

```
void MaClasse::maMethode() // Ne pas remettre "static" dans
l'implémentation
{
    cout << "Bonjour !" << endl;
}
```

Ensuite, dans le main, la méthode statique s'appelle comme ceci :

Code : C++

```
int main()
{
    MaClasse::maMethode();

    return 0;
}
```




Mais... on n'a pas créé d'objet de type `MaClasse` et on appelle la méthode quand même ? C'est quoi ce bazar ?

C'est justement ça la particularité des méthodes statiques. Pour les utiliser, pas besoin de créer un objet. Il suffit juste de faire précéder le nom de la méthode par le nom de la classe suivi de deux deux-points.

D'où le : `MaClasse::maMethode()`;

Cette méthode, comme je vous le disais, ne peut pas accéder aux attributs de la classe. C'est vraiment une bête fonction, mais rangée dans une classe. Ça permet de regrouper les fonctions dans des classes, par thème, et aussi d'éviter des conflits de nom.

Quelques exemples de l'utilité des méthodes statiques

Les méthodes statiques peuvent vous paraître un tantinet stupides. En effet, à quoi bon avoir inventé le modèle objet si c'est pour autoriser les gens à créer de bêtes "fonctions" regroupées dans des classes ?

La réponse, c'est qu'on a toujours besoin d'utiliser de "bêtes" fonctions même en modèle objet, mais pour être un peu cohérent on les regroupe dans des classes en précisant qu'elles sont statiques.

Il y a en effet des fonctions qui ne nécessitent pas de créer un objet, pour lesquelles ça n'aurait pas de sens. Des exemples ?

- Il existe dans la bibliothèque Qt une classe **QDate** qui permet de manipuler des dates. On peut comparer des dates entre elles (surcharge d'opérateur) etc etc. Cette classe propose aussi un certain nombre de méthodes statiques, comme `currentDate()` qui renvoie la date actuelle. Pas besoin de créer un objet pour avoir cette information ! Il suffit donc de taper `QDate::currentDate()` pour récupérer la date actuelle 😊
- Toujours avec Qt, la classe **QDir**, qui permet de manipuler les dossiers du disque dur, propose quelques méthodes statiques. Par exemple, on trouve `QDir::drives()` qui renvoie la liste des disques présents sur l'ordinateur (par exemple "C:\", "D:\", etc). Là encore, ça n'aurait pas eu d'intérêt d'instancier un objet à partir de la classe car ce sont des *informations générales*.
- etc etc.

Mmmh mais c'est que ça donne envie de travailler avec Qt tout ça 😊

Les attributs statiques

Il existe aussi ce qu'on appelle des **attributs statiques**.

Tout comme les méthodes statiques, les attributs statiques appartiennent à la classe et non aux objets créés à partir de la classe.

Créer un attribut statique dans une classe

C'est assez simple en fait : il suffit de rajouter le mot-clé **static** au début de la ligne.

Un attribut **static**, bien qu'il soit accessible de l'extérieur, peut très bien être déclaré *private* ou *protected*. Appelez ça une exception, car c'en est bien une. 😊

Exemple :

Code : C++

```
class MaClasse
{
    public:
        MaClasse();

    private:
        static int monAttribut;
};
```

Sauf qu'on ne peut pas initialiser l'attribut statique ici. Il faut le faire dans l'espace global, c'est-à-dire en dehors de toute classe ou fonction, en dehors du main notamment.

Code : C++

```
// Initialiser l'attribut en dehors de toute fonction ou classe  
(espace global)  
int MaClasse::monAttribut = 5;
```



Cette ligne se met généralement dans le fichier .cpp de la classe.

Un attribut déclaré comme statique se comporte comme une variable globale, c'est-à-dire une variable accessible partout dans le code.



Il est très tentant de déclarer des attributs statiques pour pouvoir accéder partout à ces variables sans avoir à les passer en argument de fonctions par exemple. C'est généralement une mauvaise chose. Cela pose des gros problèmes de maintenance. En effet, comme l'attribut est accessible depuis partout, comment savoir à quel moment il va être modifié ? Imaginez un programme avec des centaines de fichiers dans lequel vous devez chercher l'endroit qui modifie cet attribut ! C'est impossible.

N'utilisez donc des attributs statiques que si vous en avez réellement besoin. 🧑🔧

Une des utilisations les plus courantes des attributs statiques est la création d'un compteur d'instance. Il arrive parfois que l'on ait besoin de connaître combien d'objets d'une certaine classe ont été créés.

Pour y arriver, on crée alors un attribut statique `compteur` que l'on initialise à zéro. On incrémente ensuite ce compteur dans les constructeurs de la classe et bien sûr on le décrémente dans le destructeur. Et comme toujours, il nous faut respecter l'encapsulation (eh oui, on ne veut pas que tout le monde puisse changer le nombre d'objets sans en créer ou en détruire !). Il nous faut donc mettre notre attribut dans la partie privée de la classe et ajouter un accesseur. Cet accesseur est bien sûr une méthode statique ! 😊

Code : C++ - Un compteur d'instance

```
class Personnage  
{  
    public:  
        Personnage(string nom);  
        // Plein de méthodes...  
        ~Personnage();  
  
        static int nombreInstances();  
  
    private:  
        string m_nom;  
        static int compteur;  
}
```

Et tout se passe ensuite dans le .cpp correspondant.

Code : C++

```
int Personnage::compteur = 0; //On initialise à 0 notre compteur  
  
Personnage::Personnage(string nom)
```

```

        :m_nom(nom)
    {
        ++compteur;    //Quand on crée un personnage, on ajoute 1 au
compteur;
    }

    Personnage::~~Personnage()
    {
        --compteur;    //Et on enlève 1 au compteur lors de la
destruction
    }

    int Personnage::nombreInstances()
    {
        return compteur;
    }

```

On peut alors à tout instant connaître le nombre de personnages présents dans le jeu en consultant la valeur de l'attribut `Personnage::compteur`, c'est-à-dire en appelant la méthode `nombreInstances()`.

Code : C++

```

int main()
{
    //On crée deux personnages
    Personnage goliath("Goliath le tenebreux");
    Personnage lancelet("Lancelot le preux");

    //Et on consulte notre compteur
    cout << "Il y a actuellement " << Personnage::nombreInstances()
<< " personnages en jeu." << endl;
    return 0;
}

```

Simple et efficace non ? Vous verrez dans la suite d'autres exemples d'attributs statiques. Ce n'est pas ça qui manque en C++. 😊

L'amitié

Vous savez créer des classes mères, des classes filles, des classes petites-filles, etc. Un vrai arbre généalogique en quelque sorte. Mais en POO, comme dans la vie, il n'y a pas que la famille, il y a aussi les amis.

Qu'est-ce que l'amitié ?

L'amitié dans les langages orientés objet est le fait de donner un accès complet aux éléments d'une classe.

Donc si je déclare une fonction `f` amie de la classe `A`, la fonction `f` pourra modifier les attributs de la classe `A` même si les attributs sont privés ou protégés. La fonction `f` pourra également utiliser les fonctions privées et protégées de la classe `A`.

On dit alors que la **fonction `f` est amie de la classe `A`**.

En déclarant une fonction amie d'une classe, on casse complètement l'encapsulation de la classe puisqu'un être externe à la classe pourra modifier ce qu'il y a dedans. **Il ne faut donc pas abuser de l'amitié.** 🤖

Je vous ai expliqué dès le début que l'encapsulation était l'élément le plus important en POO et voilà que je vous présente un moyen de détourner ce concept. Je suis d'accord avec vous, c'est assez paradoxal. Pourtant, utiliser à *bon escient* l'amitié peut renforcer l'encapsulation. Voyons comment !

Retour sur la classe Duree

Pour vous présenter la surcharge des opérateurs, j'ai utilisé la classe `Duree` dont le but était de représenter la notion d'intervalle de temps. Voici le prototype de la classe :

Code : C++ - Prototype de la classe `Duree`

```
class Duree
{
    public:

        Duree(int heures = 0, int minutes = 0, int secondes = 0);
        void affiche(ostream& out) const;    //Permet d'écrire la durée
        dans un flux

    private:

        int m_heures;
        int m_minutes;
        int m_secondes;
};

//Surcharge de l'opérateur << pour l'écriture dans les flux
//Utilise la méthode affiche() de Duree
ostream &operator<<( ostream &out, Duree const& duree );
```

Je ne vous ai mis que l'essentiel. Il y avait bien plus d'opérateurs déclarés à la fin du chapitre. Ce qui va nous intéresser, c'est la surcharge de l'opérateur d'injection dans les flux. Voici ce que nous avons écrit :

Code : C++ - Surcharge de l'opérateur <<

```
ostream &operator<<( ostream &out, Duree const& duree )
{
    duree.affiche(out) ;
    return out;
}
```

Et c'est très souvent la meilleure solution ! Mais pas toujours... En effet, en faisant ça, vous avez besoin d'écrire une méthode `affiche()` dans la classe. C'est-à-dire, que votre classe va fournir un service supplémentaire. Vous allez ajouter un levier en plus en surface de votre classe.



Sauf que ce levier n'est destiné qu'à l'opérateur << et pas au reste du monde. Il y a donc une méthode dans la classe qui d'une certaine manière ne sert à rien pour un utilisateur normal.

Dans ce cas, cela ne porte pas vraiment à conséquence. Si quelqu'un utilise la méthode `affiche()`, alors rien de dangereux pour l'objet ne se passe. Il s'écrit juste dans la console ou dans un fichier. Mais dans d'autres cas, il pourrait être dangereux d'avoir une méthode qu'il ne faut surtout pas utiliser.

C'est comme dans les laboratoires, si vous avez un gros bouton rouge avec un écriteau indiquant "Ne surtout pas appuyer", vous pouvez être sûr que quelqu'un va, un jour, faire l'erreur d'appuyer dessus. 🤪

Le mieux serait donc de ne pas laisser apparaître ce levier en surface de notre cube-objet. Ce qui revient à mettre la méthode `affiche()` dans la partie privée de la classe.

Code : C++ - Prototype de la classe *Duree*

```
class Duree
{
    public:

        Duree(int heures = 0, int minutes = 0, int secondes = 0);

    private:

        void affiche(ostream& out) const;    //Permet d'écrire la durée
        dans un flux

        int m_heures;
        int m_minutes;
        int m_secondes;
};
```

En faisant cela, plus de risque d'appeler la méthode par erreur. Par contre, l'opérateur << ne peut plus, lui non plus, l'utiliser. C'est là que l'amitié intervient. Si l'opérateur << est déclaré ami de la classe *Duree*, il aura accès à la partie privée de la classe et par conséquent à la méthode `affiche()`.

Déclarer une fonction amie d'une classe





Interro surprise d'anglais. Comment dit-on « ami » en anglais ?

Friend, exactement ! Et comme les créateurs du C++ ne voulaient pas se casser la tête avec les noms compliqués, ils ont pris **friend** comme mot-clé pour l'amitié. D'ailleurs si vous tapez ce mot dans votre IDE, il devrait s'écrire d'une couleur différente. C'est bien la preuve. 😊

Pour déclarer une fonction amie d'une classe, on utilise la syntaxe suivante :

Code : C++

```
friend std::ostream& operator<< (std::ostream& flux, Duree const&
duree);
```

On écrit **friend** suivi du prototype de la fonction. Et on place le tout à l'intérieur de la classe.

Code : C++ - Prototype de la classe Duree

```
class Duree
{
    public:

        Duree(int heures = 0, int minutes = 0, int secondes = 0);

    private:

        void affiche(ostream& out) const;    //Permet d'écrire la durée
        dans un flux

        int m_heures;
        int m_minutes;
        int m_secondes;

        friend std::ostream& operator<< (std::ostream& flux, Duree
const& duree);
};
```



Vous pouvez mettre le prototype de la fonction dans la partie publique, protégée ou privée de la classe, cela n'a aucune importance.

Notre opérateur << a maintenant accès à tout ce qui se trouve dans la classe `Duree` sans aucune restriction. Il peut donc en particulier utiliser la méthode `affiche()`, comme précédemment. Sauf que désormais, c'est le seul élément hors de la classe qui peut utiliser cette méthode.

On peut utiliser la même astuce pour les opérateurs `==` et `<`. En les déclarant comme amis de la classe `Duree`, ces fonctions pourront accéder directement aux attributs et l'on peut alors supprimer les méthodes `estPlusPetitQue()` et `estEgal()`. Je vous laisse essayer ... 😊

L'amitié et la responsabilité

Être l'ami de quelqu'un a certaines conséquences en matière de savoir-vivre. Je présume que vous n'allez pas chez vos amis à 3h du matin pour saccager leur jardin pendant leur sommeil.

En C++, l'amitié implique également que la fonction amie ne viendra pas détruire la classe et ne viendra pas non plus saccager les attributs de la classe. Si vous avez besoin d'une fonction qui doit modifier grandement le contenu d'une classe, alors faites plutôt une fonction membre de la classe.

Vos programmes devraient respecter les deux règles suivantes :

- Une fonction amie ne devrait, en principe, pas modifier l'instance de la classe.
- Utilisez les fonctions amies que si vous ne pouvez pas faire autrement.

Cette deuxième règle est très importante. Si vous ne la respectez pas, alors autant arrêter la POO, car le concept de classe perd tout son sens.

Après avoir emmagasiné toutes ces connaissances, nous allons pratiquer un peu avec la bibliothèque Qt et créer des fenêtres !



Partie 3 : [Pratique] Créez vos propres fenêtres avec Qt

Vous l'avez compris en lisant la partie II : la POO, ce n'est pas évident à maîtriser au début, mais ça apporte un nombre important d'avantages : le code est plus facile à réutiliser, à améliorer, et... quand on utilise une bibliothèque là c'est carrément le pied 😊

Le but de la partie III est entièrement de pratiquer, pratiquer, pratiquer. Vous n'apprendrez pas de nouvelles notions théoriques ici, mais par contre vous allez apprendre à maîtriser le C++ par la pratique, et ça c'est important.



Qt est une bibliothèque C++ très complète qui vous permet notamment de créer vos propres fenêtres, que vous soyez sous Windows, Linux ou Mac OS. Tout ce que nous allons faire sera très concret : ouverture de fenêtres, ajout de boutons, création de menus, de listes déroulantes... bref que des choses motivantes ! 😊

Introduction à Qt

Les amis, le temps n'est plus aux bavardages mais au **concret** !

Vous trouverez difficilement plus concret que cette partie du cours 😊



Pour bien pouvoir comprendre cette partie, il est vital que vous ayez lu et compris le début de ce cours.

Si certaines notions de la programmation orientée objet vous sont encore un peu obscures, n'hésitez pas à faire un tour à nouveau sur les chapitres correspondants. Au pire des cas, si vraiment ça ne rentre pas, vous pouvez quand même lire cette partie, vous aurez peut-être un déclic en pratiquant. 😊

Nous commencerons dans un premier temps par découvrir ce qu'est Qt concrètement, ce que cette bibliothèque permet de faire, et quelles sont aussi les alternatives qui existent (car il n'y a pas qu'avec Qt qu'on peut créer des fenêtres !).

Nous verrons ensuite comment installer et configurer Qt.

Préparez-vous bien, parce que dès le chapitre suivant on attaque dare-dare !

Dis papa, comment on fait des fenêtres ?

Voilà une question que vous vous êtes tous déjà posés, j'en suis sûr ! J'en mettrais même ma main à couper (*et j'y tiens à ma main, c'est vous dire* 😊).



Alors alors, c'est comment qu'on programme des fenêtres ? 😊

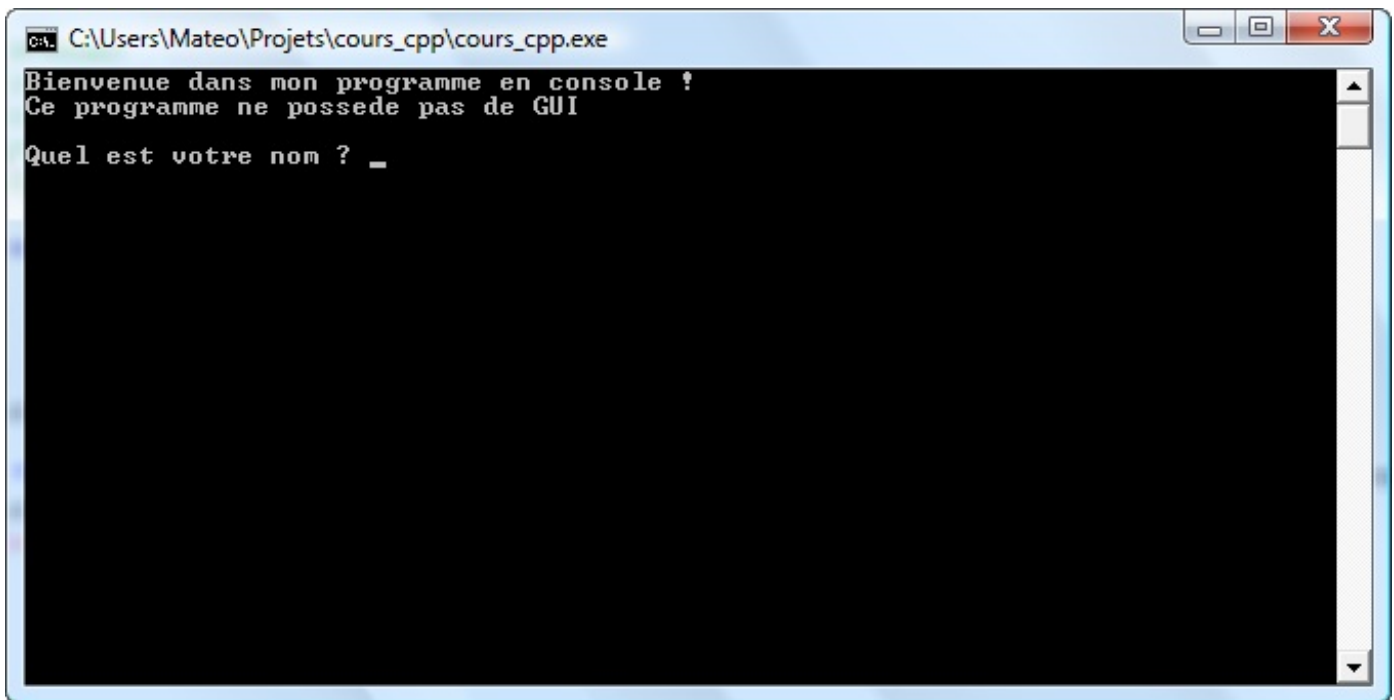
Douuuusement, pas d'impatience. Si vous allez trop vite vous risquez de brûler des étapes et de vous retrouver bloqué après, alors allez-y progressivement et dans l'ordre en écoutant bien tout ce que j'ai à vous dire.

Un mot de vocabulaire à connaître : GUI

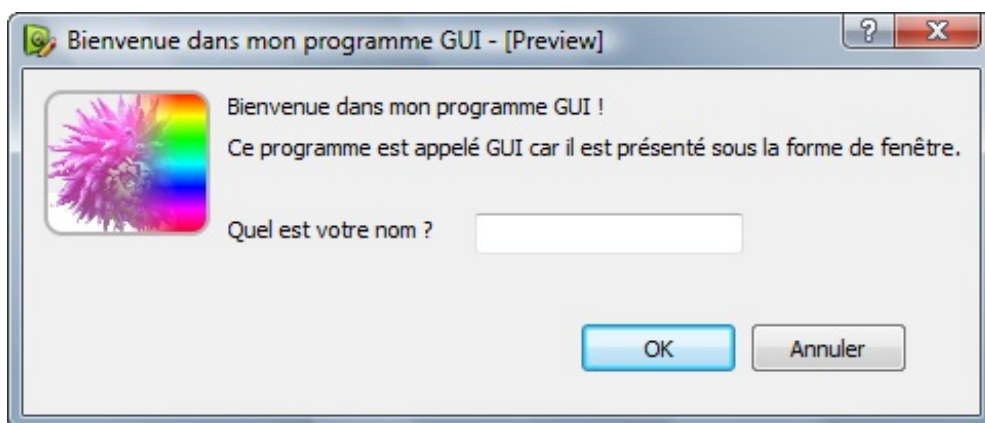
Avant d'aller plus loin, je voudrais vous faire apprendre ce petit mot de vocabulaire car je vais le réutiliser tout au long de cette partie **GUI** (prononcez "Goui").

C'est l'abréviation de **Graphical User Interface**, soit "Interface utilisateur graphique". Ça désigne tout ce qu'on appelle grossièrement "Programme avec des fenêtres".

Pour bien que vous puissiez comparer, voici un programme sans GUI (en console) et un programme GUI :



Programme sans GUI (console)



Programme GUI, ici sous Windows

Les différents moyens de créer des GUI

Chaque système d'exploitation (Windows, Mac OS X, Linux...) propose au moins un moyen de créer des fenêtres... le problème, c'est justement que ce moyen n'est en général pas *portable*, c'est-à-dire que votre programme créé uniquement pour Windows ne pourra marcher que sous Windows et pas ailleurs.

On a grosso modo 2 types de choix :

- Soit on écrit son application **spécialement pour l'OS** qu'on veut, mais le programme ne sera pas portable.
- Soit on utilise une bibliothèque **qui s'adapte à tous les OS**, c'est-à-dire une bibliothèque multi-plateforme.

La deuxième solution est en générale la meilleure car c'est la plus souple. C'est d'ailleurs celle que nous allons choisir pour que personne ne se sente abandonné.

Histoire d'être suffisamment complet quand même, je vais dans un premier temps vous parler des bibliothèques propres aux principaux OS pour que vous connaissiez au moins leurs noms.

Ensuite, nous verrons quelles sont les principales bibliothèques multi-plateforme.

Les bibliothèques propres aux OS

Chaque OS propose au moins une bibliothèque qui permet de créer des fenêtres. Le défaut de cette méthode est qu'en général cette bibliothèque ne marche que pour l'OS pour lequel elle a été créée. Ainsi, si vous utilisez la bibliothèque de Windows, votre programme ne marchera que sous Windows.

- **Sous Windows** : on dispose du framework .NET. C'est un ensemble très complet de bibliothèques utilisables en C++, C#, Visual Basic... Le langage de prédilection pour travailler avec .NET est C#. A noter que .NET peut aussi être utilisé sous Linux (avec quelques limitations) grâce au projet Mono.
En somme, .NET est un vrai couteau suisse pour développer sous Windows et on peut aussi faire fonctionner les programmes sous Linux à quelques exceptions près.
- **Sous Mac OS X** : la bibliothèque de prédilection s'appelle Cocoa. On l'utilise en général en langage "Objective C". C'est une bibliothèque orientée objet.
- **Sous Linux** : tous les environnements de bureaux (appelés WM, Windows Managers) reposent sur X, la base des interfaces graphiques de Linux. X propose une bibliothèque appelée Xlib, mais on programme rarement en Xlib sous Linux. On préfère utiliser une bibliothèque plus simple d'utilisation et multi-plateforme comme GTK+ (sous Gnome) ou Qt (sous KDE).

Comme vous le voyez, il y a en gros une bibliothèque "de base" pour chaque OS. Certaines, comme Cocoa, ne fonctionnent que pour le système d'exploitation pour lequel elles ont été prévues. Il est généralement conseillé d'utiliser une bibliothèque multi-plateforme si vous comptez distribuer votre programme à un grand nombre de personnes.

Les bibliothèques multi-plateforme

Les avantages d'utiliser une bibliothèque multi-plateforme sont nombreux. Même si vous voulez créer des programmes pour Windows et que vous n'en avez rien à faire de Linux et Mac OS, oui oui 😊

- Tout d'abord, elles simplifient grandement la création d'une fenêtre. Il faut beaucoup moins de lignes de code pour ouvrir une "simple" fenêtre.
- Ensuite, elles uniformisent le tout, elles forment un ensemble cohérent qui fait qu'il est facile de s'y retrouver. Les noms des fonctions et des classes sont choisis de manière logique de manière à vous aider autant que possible.
- Enfin, elles font abstraction du système d'exploitation mais aussi de la version du système. Cela veut dire que si demain Cocoa cesse d'être utilisable sous Mac OS X, votre application continuera à fonctionner car la bibliothèque multi-plateforme s'adaptera aux changements.

Bref, choisir une bibliothèque multi-plateforme, ce n'est pas seulement pour que le programme marche partout, mais aussi pour être sûr qu'il marchera tout le temps et pour avoir un certain confort en programmant.

Voici quelques-unes des principales bibliothèques multi-plateforme à connaître, au moins de nom :

- **.NET** (prononcez "Dot Net") : développé par Microsoft pour succéder à la vieillissante API Win32. On l'utilise souvent en langage C#. On peut néanmoins utiliser .NET dans une multitude d'autres langages dont le C++. .NET est portable car Microsoft a expliqué son fonctionnement. Ainsi, on peut utiliser un programme écrit en .NET sous Linux avec Mono. Pour le moment néanmoins, .NET est principalement utilisé sous Windows.
- **GTK+** : une des plus importantes bibliothèques utilisées sous Linux. Elle est portable, c'est-à-dire utilisable sous Linux, Mac OS et Windows. GTK+ est utilisable en C. Néanmoins, il existe une version C++ appelée GTKmm (on parle de *wrapper*, ou encore de *surcouche*).
GTK+ est la bibliothèque de prédilection pour ceux qui écrivent des applications pour Gnome sous Linux, mais elle fonctionne aussi sous KDE.
C'est la bibliothèque utilisée par Firefox par exemple, pour ne citer que lui.
- **Qt** : bon je ne vous la présente pas trop longuement ici car tout ce chapitre est là pour ça. 😊
Sachez néanmoins que Qt est très utilisée sous Linux aussi, en particulier sous l'environnement de bureau KDE.
- **wxWidgets** : une bibliothèque objet très complète elle aussi, comparable en gros à Qt. Sa licence est très semblable à celle de Qt (elle vous autorise à créer des programmes propriétaires). Néanmoins, j'ai choisi quand même de vous montrer Qt car cette bibliothèque est plus facile à prendre en main au début. Sachez qu'une fois qu'on l'a prise en main, wxWidgets n'est pas beaucoup plus compliquée que Qt.
wxWidgets est la bibliothèque utilisée pour réaliser le GUI de l'IDE Code::Blocks.
- **FLTK** : contrairement à toutes les bibliothèques "poids lourd" précédentes, FLTK se veut légère. C'est une petite

bibliothèque dédiée uniquement à la création d'interfaces graphiques multi-plateforme.

Comme vous le voyez, j'ai dû faire un choix parmi tout ça. 😊

C'est la qualité de la bibliothèque Qt et de sa documentation qui m'a convaincu de vous la présenter.

Présentation de Qt

Vous l'avez compris, Qt est une **bibliothèque** multi-plateforme pour créer des GUI (programme sous forme de fenêtre).

Qt est écrite en C++ et est faite pour être utilisée à la base en C++, mais il est aujourd'hui possible de l'utiliser dans d'autres langages comme Java, Python, etc.

Plus fort qu'une bibliothèque : un framework

Qt est en fait... bien plus qu'une bibliothèque. C'est un ensemble de bibliothèques. Le tout est tellement énorme qu'on parle d'ailleurs plutôt de **framework** : cela signifie que vous avez à votre disposition un ensemble d'outils pour développer vos programmes plus efficacement.

Qu'on ne s'y trompe pas : Qt est à la base faite pour créer des fenêtres, c'est en quelque sorte sa fonction centrale. Mais ce serait dommage de limiter Qt à ça.

Qt est donc constituée d'un ensemble de bibliothèques, appelées "modules". On peut y trouver entre autres ces fonctionnalités :



- **Module GUI** : c'est toute la partie création de fenêtres. Nous nous concentrerons surtout sur le module GUI dans ce cours.
- **Module OpenGL** : Qt peut ouvrir une fenêtre contenant de la 3D gérée par OpenGL.
- **Module de dessin** : pour tous ceux qui voudraient dessiner dans leur fenêtre (en 2D), le module de dessin est très complet !
- **Module réseau** : Qt fournit une batterie d'outils pour accéder au réseau, que ce soit pour créer un logiciel de Chat, un client FTP, un client Bittorrent, un lecteur de flux RSS...
- **Module SVG** : possibilité de créer des images et animations vectorielles, à la manière de Flash.
- **Module de script** : Qt supporte le Javascript (ou ECMAScript), que vous pouvez réutiliser dans vos applications pour ajouter des fonctionnalités, sous forme de plugins par exemple.
- **Module XML** : pour ceux qui connaissent le XML, c'est un moyen très pratique d'échanger des données avec des fichiers formés à l'aide de balises, un peu comme le XHTML.
- **Module SQL** : permet un accès aux bases de données (MySQL, Oracle, PostgreSQL...).

Que les choses soient claires : Qt n'est pas gros, Qt est **énorme**, et il ne faut pas compter sur un tutoriel pour vous expliquer tout ce qu'il y a à savoir sur Qt. Je vais vous montrer beaucoup de ses possibilités mais on ne pourra jamais tout voir. On se concentrera surtout sur la partie GUI.

Pour ceux qui veulent aller plus loin, il faudra lire la [documentation officielle](#) (uniquement en anglais, comme toutes les documentations pour les programmeurs de toute façon). Cette documentation est très bien faite, elle détaille toutes les fonctionnalités de Qt, même les plus récentes.

Sachez d'ailleurs que j'ai choisi Qt en grande partie parce que sa documentation est très bien faite et facile à utiliser. Vous aurez donc intérêt à vous en servir. 😊

Si vous êtes perdu ne vous en faites pas, je vous expliquerai dans un prochain chapitre comment on fait pour "lire" et naviguer dans une telle documentation.

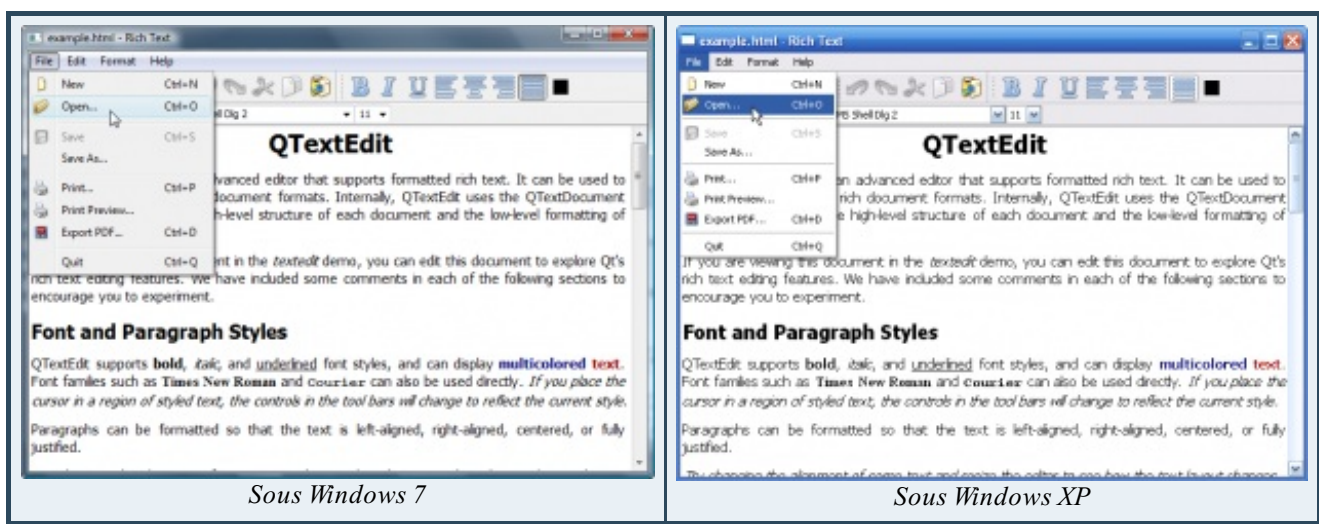
Qt est multiplateforme

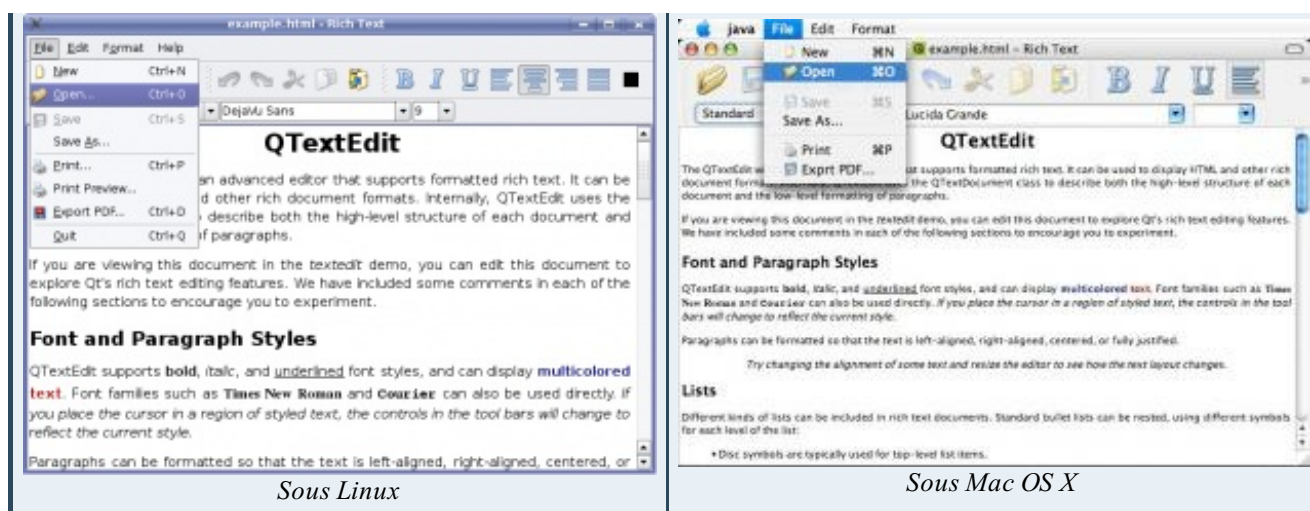
Qt est un **framework multiplateforme**. Je le sais je me répète, mais c'est important de l'avoir bien compris. Tenez, d'ailleurs voilà un schéma qui illustre le fonctionnement de Qt :



Grâce à cette technique, les fenêtres que vous codez ont un "look" adapté à chaque OS. Vous codez pour Qt, et Qt traduit les instructions pour l'OS. Les utilisateurs de vos programmes n'y verront que du feu et ne sauront pas que vous utilisez Qt (de toute manière ils s'en moquent 😊).

Voici une démonstration de ce que je viens de vous dire. Vous avez ci-dessous le même programme, donc la même fenêtre créée avec Qt, mais sous différents OS. Vous allez voir que Qt s'adapte à chaque fois :





Tout ce que vous avez à faire pour produire le même résultat, c'est recompiler votre programme sous chacun de ces OS. Par exemple, vous avez développé votre programme sous Windows, très bien, mais les .exe n'existent pas sous Linux. Il vous suffit simplement de recompiler votre programme sous Linux et c'est bon, vous avez une version Linux !



On est obligé de recompiler pour chacun des OS ?

Oui, ça vous permet de créer des programmes binaires adaptés à chaque OS qui tournent à pleine vitesse.

On ne va toutefois pas se préoccuper de compiler sous chacun des OS maintenant, on va déjà le faire pour votre OS ça sera bien



Pour information, d'autres langages de programmation comme Java et Python ne nécessitent pas de recompilation car le terme "compilation" n'existe pas vraiment sous ces langages. Cela fait que les programmes sont un peu plus lents, mais ils s'adaptent automatiquement partout.

L'avantage du C++ par rapport à ces langages est donc sa rapidité (bien que la différence se sente de moins en moins, sauf pour les jeux vidéo qui ont besoin de rapidité et qui sont donc majoritairement codés en C++).

L'histoire de Qt

Bon, ne comptez pas sur moi pour vous faire un historique long et chiant sur Qt, mais je pense qu'un tout petit peu de culture générale ne peut pas vous faire de mal et vous permettra de savoir de quoi vous parlez. 😊

Qt est un framework développé initialement par la société Trolltech, qui fut racheté par Nokia par la suite.

Le développement de Qt a commencé en 1991 (ça remonte pas mal donc) et il a été dès le début utilisé par KDE, un des principaux environnements de bureau de Linux.

Qt s'écrit "Qt" et non "QT", donc avec un "t" minuscule (si vous faites l'erreur un fanatique de Qt vous égorgera probablement pour vous le rappeler 🤪)

Qt signifie "Cute" (prononcez "Quioute"), ce qui signifie "Mignonne", parce que les développeurs trouvaient que la lettre Q était jolie dans leur éditeur de texte. Oui je sais, ils sont fous ces programmeurs.

La licence de Qt

Qt est distribué sous deux licences, au choix : LGPL ou propriétaire. Celle qui nous intéresse est la licence LGPL car elle nous permet d'utiliser gratuitement Qt (et même d'avoir accès à son code source si on veut !). On peut aussi bien réaliser des programmes libres que des programmes propriétaires.

Bref, c'est vraiment l'idéal pour nous. On peut l'utiliser gratuitement et en faire usage dans des programmes libres comme dans des programmes propriétaires. 😊

Qui utilise Qt ?

Une bibliothèque comme Qt a besoin de références, c'est-à-dire d'entreprises célèbres qui l'utilisent, pour montrer son sérieux. De ce point de vue là, pas de problème. Qt est utilisée par de nombreuses entreprises que vous connaissez sûrement :

- 
Adobe
-  ARCHOS
-  BOEING
-  Google
-  NASA
-  skype™

Qt est utilisée pour réaliser de nombreux GUI, comme celui d'Adobe Photoshop Elements, de Google Earth ou encore de Skype !

Installation de Qt

Vous êtes prêts à installer Qt ?

On est parti !

Télécharger Qt

Commencez par télécharger Qt sur le [site de Qt](#).

On vous demande de choisir entre la version LGPL et la version commerciale. Comme je vous l'ai expliqué plus tôt, nous allons utiliser la version qui est sous licence LGPL.

Vous devez ensuite choisir entre :

- **Qt SDK** : la bibliothèque Qt + un ensemble d'outils pour développer avec Qt, incluant un IDE spécial appelé Qt Creator.
- **Qt Framework** : contient uniquement la bibliothèque Qt.

Je vous propose de prendre le Qt SDK, car il contient un certain nombre d'outils qui vont grandement nous simplifier la vie. 😊

Choisissez soit "Qt pour Windows: C++", "Qt pour Linux/X11: C++" ou "Qt pour Mac: C++" en fonction de votre système d'exploitation.

Dans la suite de ce chapitre, je vous présenterai l'installation du Qt SDK sous Windows.



Si vous êtes sous Linux, et notamment sous Debian ou Ubuntu, je vous recommande d'installer directement le paquet qtcreator avec la commande `apt-get install qtcreator`. La version sera peut-être légèrement plus ancienne mais l'installation sera ainsi centralisée et plus facile à gérer.

Installation sous Windows

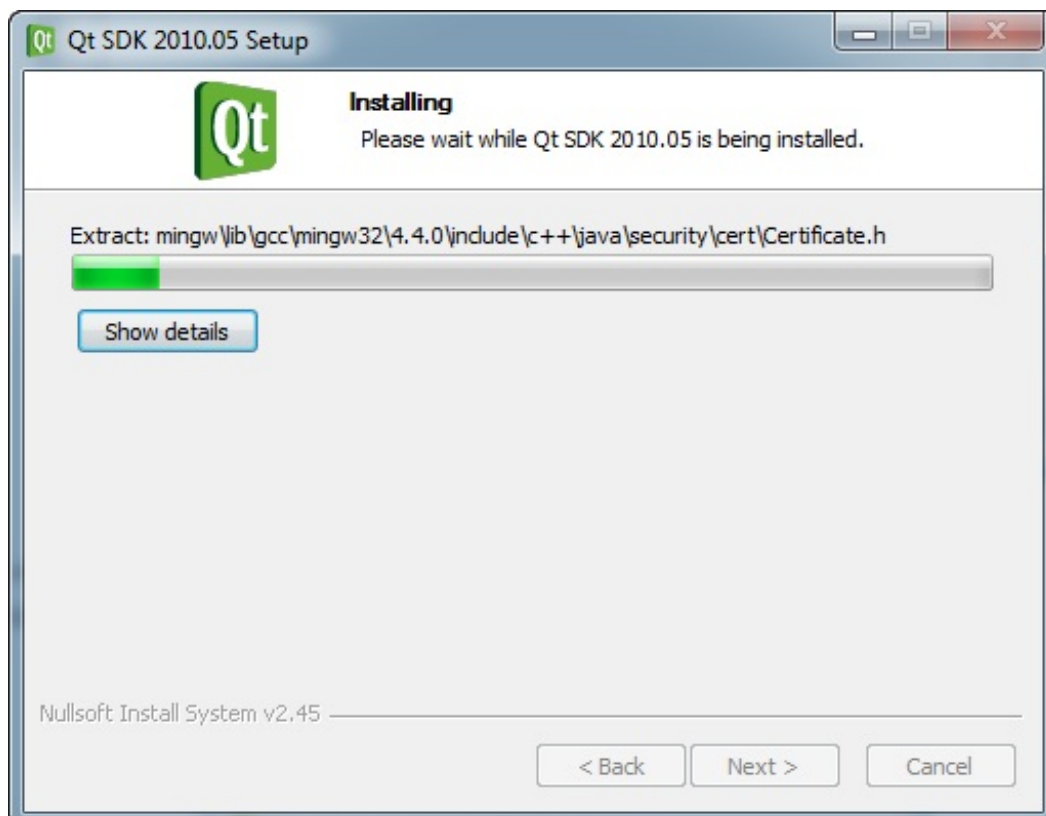
L'installation sous Windows se présente sous la forme d'un assistant d'installation classique.

Je vais vous montrer comment ça se passe pas à pas, ce n'est pas bien compliqué.

La première fenêtre est la suivante :

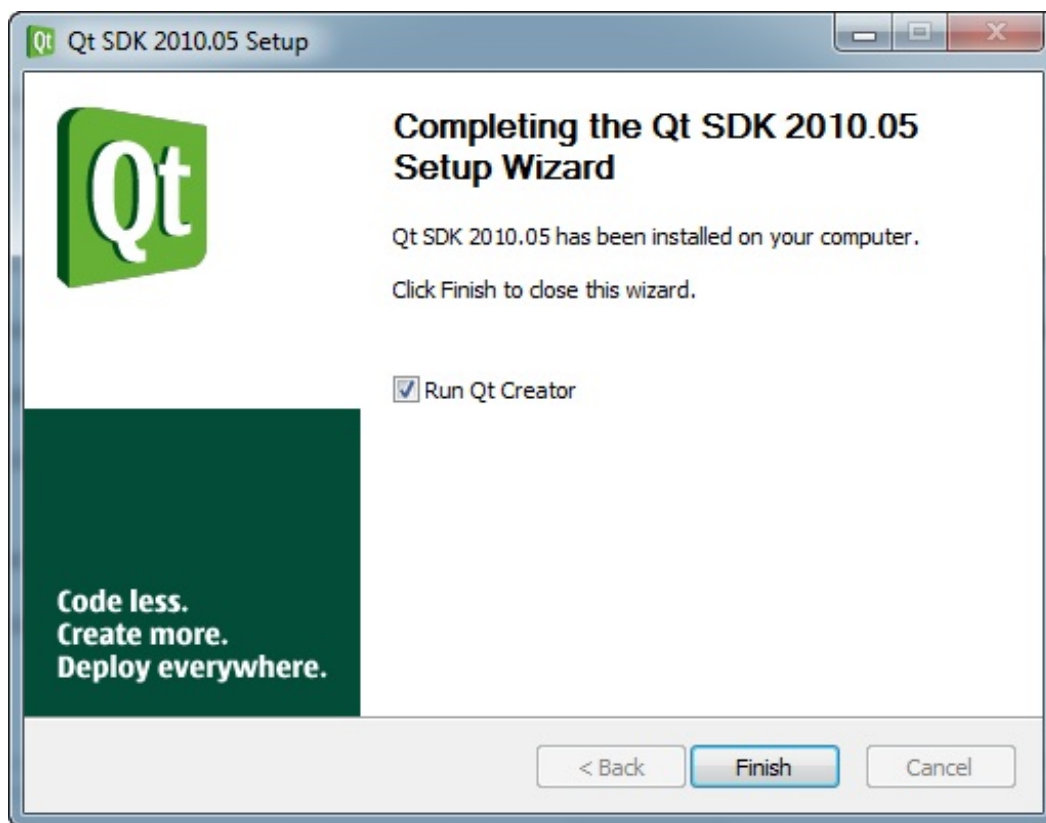


Il n'y a rien de particulier à signaler. Cliquez sur Next autant de fois que nécessaire en laissant les options par défaut. Qt s'installe ensuite (il y a beaucoup de fichiers ça peut prendre un peu de temps) :



Vous êtes à la fin ? Ouf !

On vous propose d'ouvrir le programme Qt Creator qui a été installé en plus de la bibliothèque Qt. Ce programme est un IDE spécialement optimisé pour travailler avec Qt. Je vous invite à le lancer :

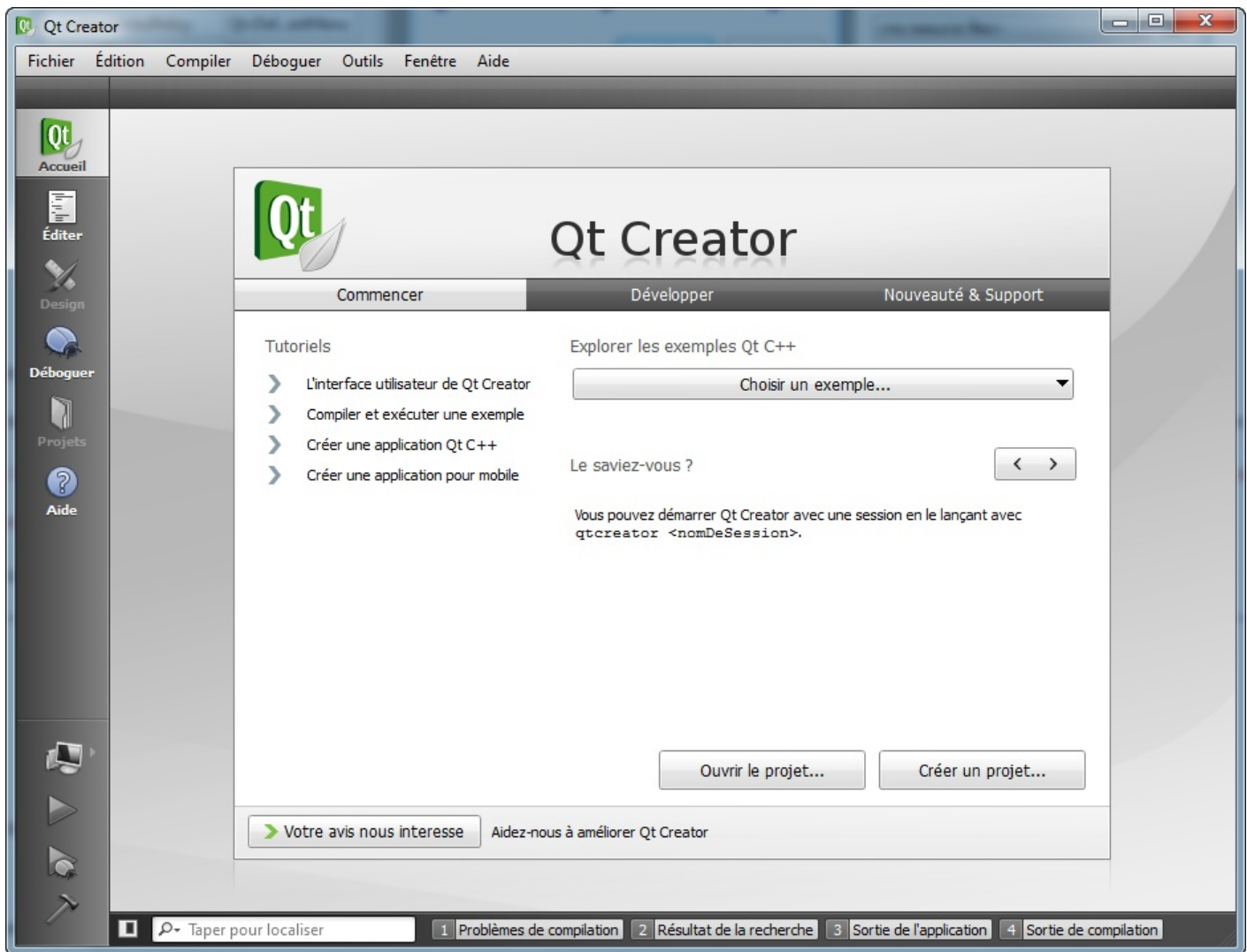


Qt Creator

Bien qu'il soit possible de développer en C++ avec Qt en utilisant notre IDE (comme Code::Blocks) je vous recommande fortement d'utiliser l'IDE Qt Creator que nous venons d'installer. Il est particulièrement optimisé pour développer avec Qt. En effet, c'est un programme tout-en-un qui comprend entre autres :

- Un IDE pour développer en C++, optimisé pour compiler des projets utilisant Qt (pas de configuration fastidieuse)
- Un éditeur de fenêtres, qui permet de dessiner facilement le contenu de ses interfaces à la souris
- Une documentation in-dis-pen-sable pour tout savoir sur Qt

Voici à quoi ressemble Qt Creator lorsque vous le lancez pour la première fois :



Comme vous le voyez, c'est un outil très propre et très bien fait. Avant que Qt Creator n'apparaisse, il fallait effectuer des configurations parfois complexes pour compiler des projets utilisant Qt. Désormais tout est transparent pour le développeur !



Dans le prochain chapitre, nous découvrirons comment utiliser Qt Creator pour développer notre premier projet Qt. Nous y compilerons notre toute première fenêtre !

Notion de GUI... OK

Présentation des bibliothèques GUI... OK

Présentation des modules de Qt... OK

Notion de framework multi-plateforme... OK

Culture générale sur Qt... OK

Téléchargement de Qt... OK

Installation de Qt... OK

Présentation des programmes livrés avec Qt... OK

- C'est bon mon commandant, ils sont parés au lancement 🤖

- Ouvrez le sas et accrochez-vous lieutenant, ça risque de bouger un peu 🧑🏻‍🚀

Compiler votre première fenêtre Qt

Bonne nouvelle, votre patience et votre persévérance vont maintenant payer. 😊

Dans ce chapitre, nous réaliserons notre premier programme utilisant Qt, et nous verrons comment ouvrir notre première fenêtre !

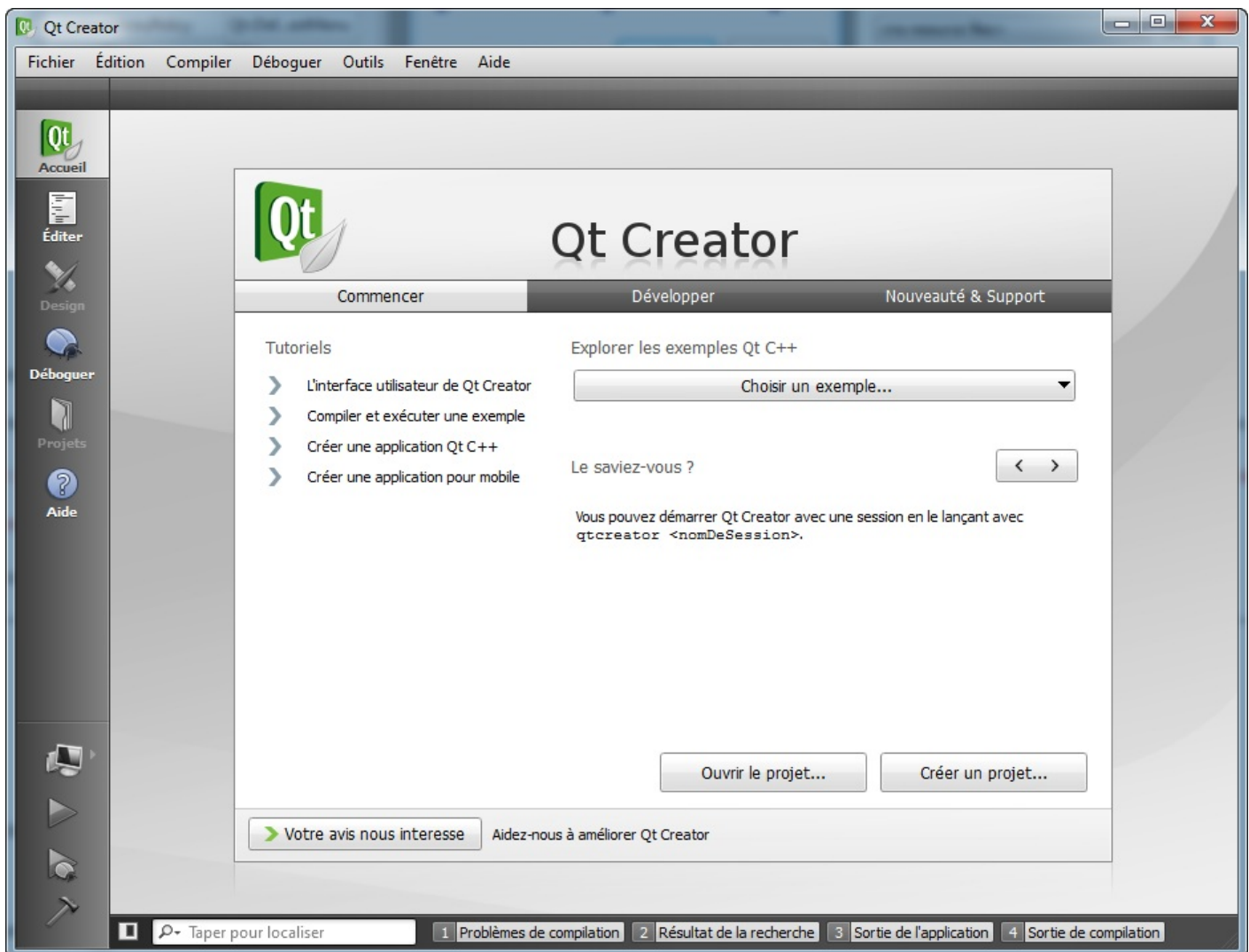
Nous allons changer d'IDE et passer de Code::Blocks à Qt Creator, qui permet de réaliser des programmes Qt beaucoup plus facilement. Je vais donc vous présenter comment utiliser Qt Creator dans ce chapitre.

Let's go !

Présentation de Qt Creator

Qt Creator, que nous avons installé dans le chapitre précédent en même temps que Qt, est l'IDE idéal pour programmer des projets Qt. Il va nous épargner des configurations fastidieuses et nous permettre de commencer à programmer en un rien de temps. 😊

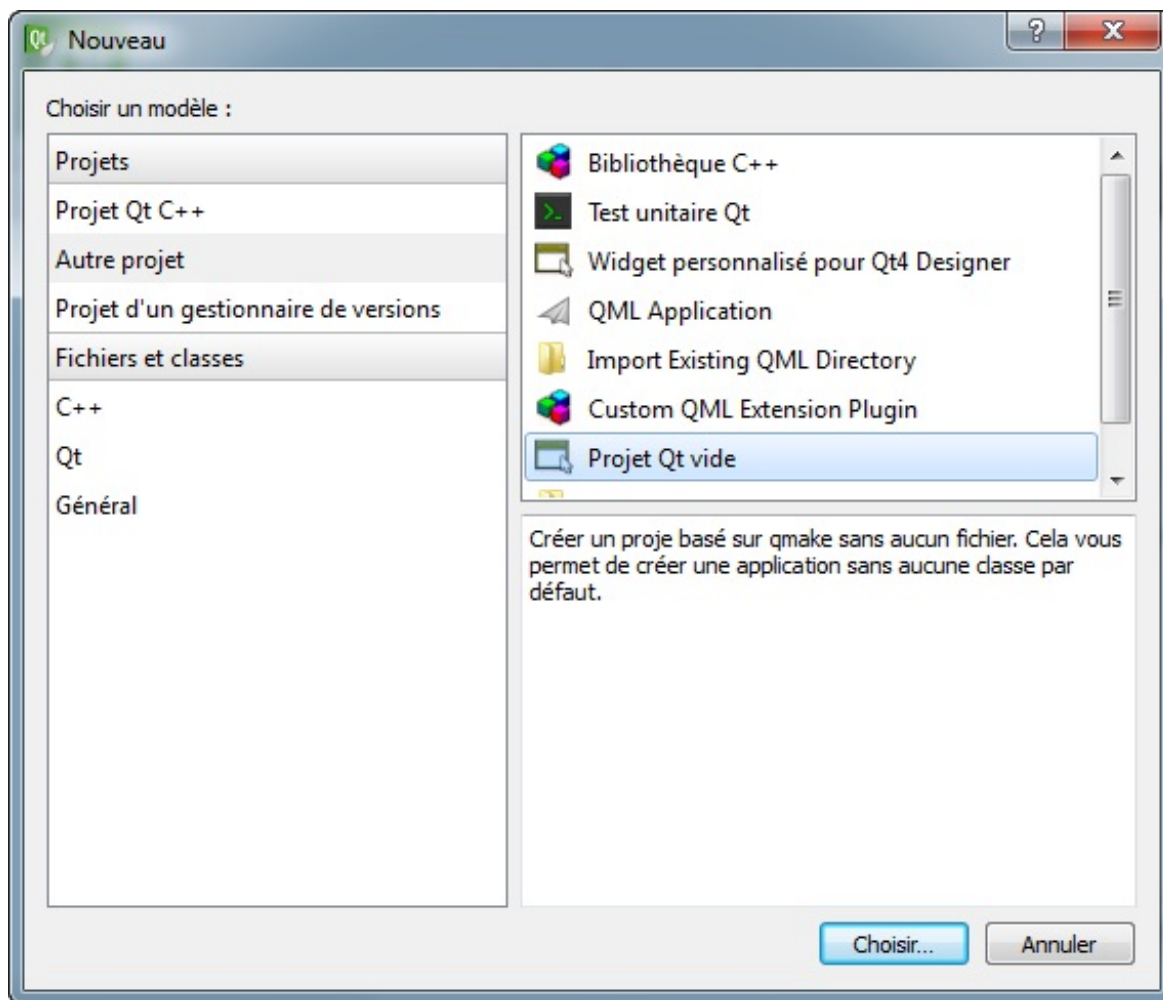
Voici la fenêtre principale du programme que nous avons déjà vue :



Création d'un projet Qt vide

Je vous propose de créer un nouveau projet en allant dans le menu **Fichier** / **Nouveau fichier ou projet**. On vous propose plusieurs choix selon le type de projet que vous souhaitez créer : application graphique pour ordinateur, application pour mobile, etc. Cependant, pour nous qui débutons, il est préférable de commencer avec un projet vide. Cela nous fera moins de fichiers à découvrir d'un seul coup. 😊

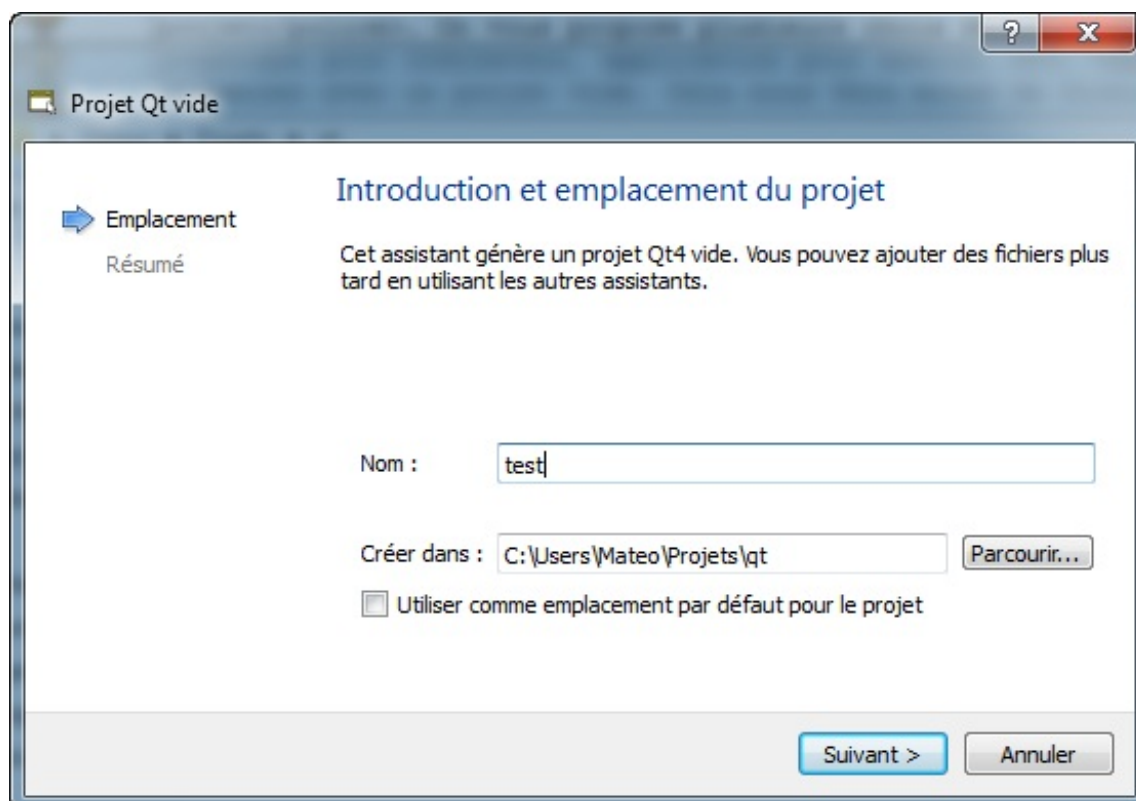
Choisissez donc les options **Autre projet** puis **Projet Qt vide** :



Nouveau projet Qt

Creator

Un assistant s'ouvre alors pour vous demander le nom du projet et l'emplacement où vous souhaitez l'enregistrer :

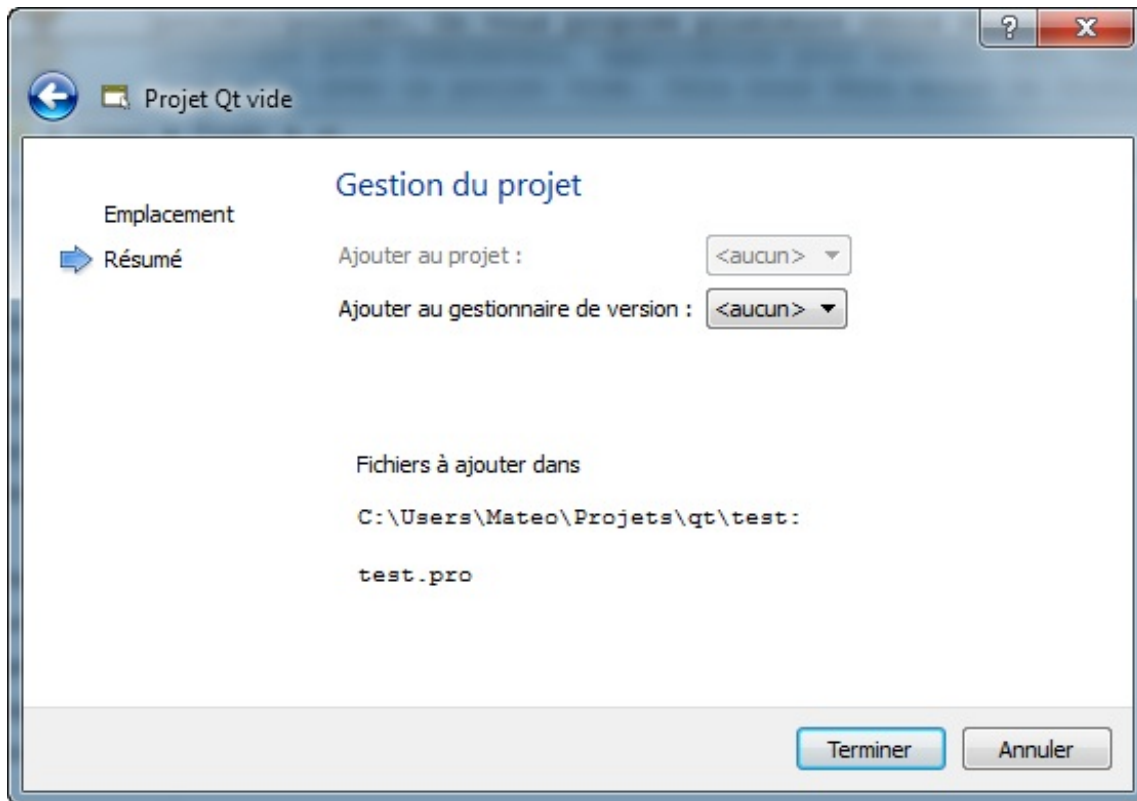


Nouveau projet

(suite)

Comme vous le voyez, je l'ai appelé "test", mais vous pouvez l'appeler comme vous voulez. 😊

La fenêtre suivante vous demande quelques informations dont vous avez peut-être moins l'habitude :



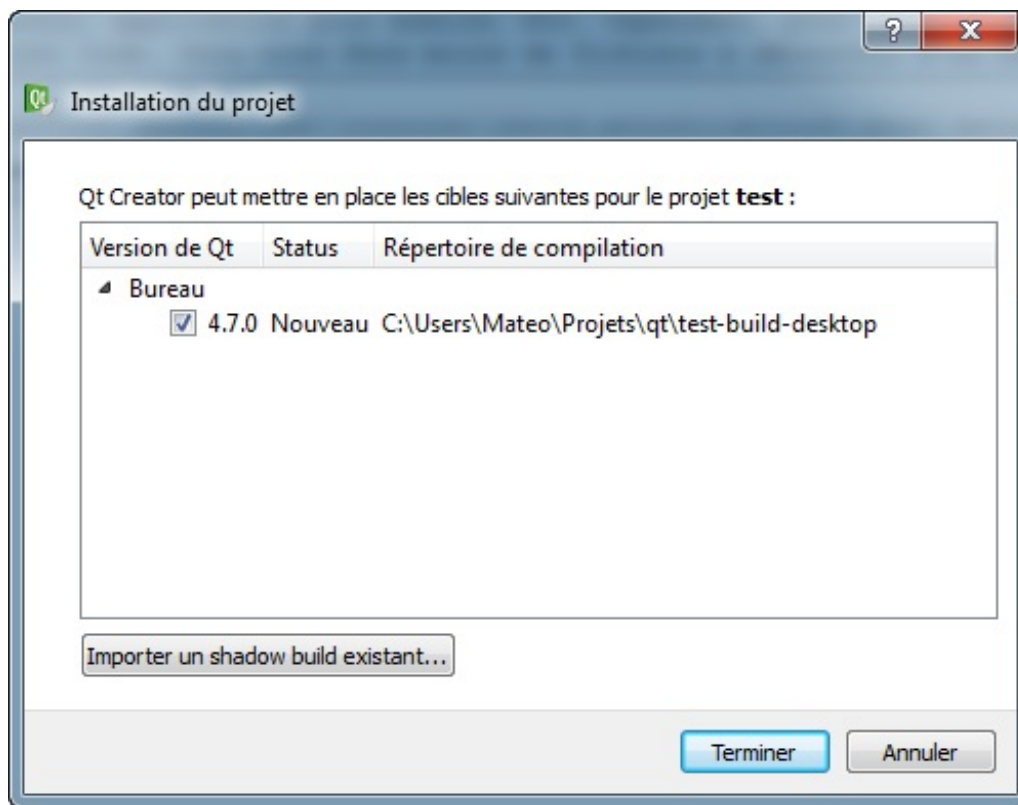
Nouveau projet

(suite)

On peut associer le projet à un gestionnaire de version (comme SVN, Git). C'est un outil très utile notamment si on travaille à plusieurs sur un code source... Mais ce n'est pas le sujet de ce chapitre. 😊

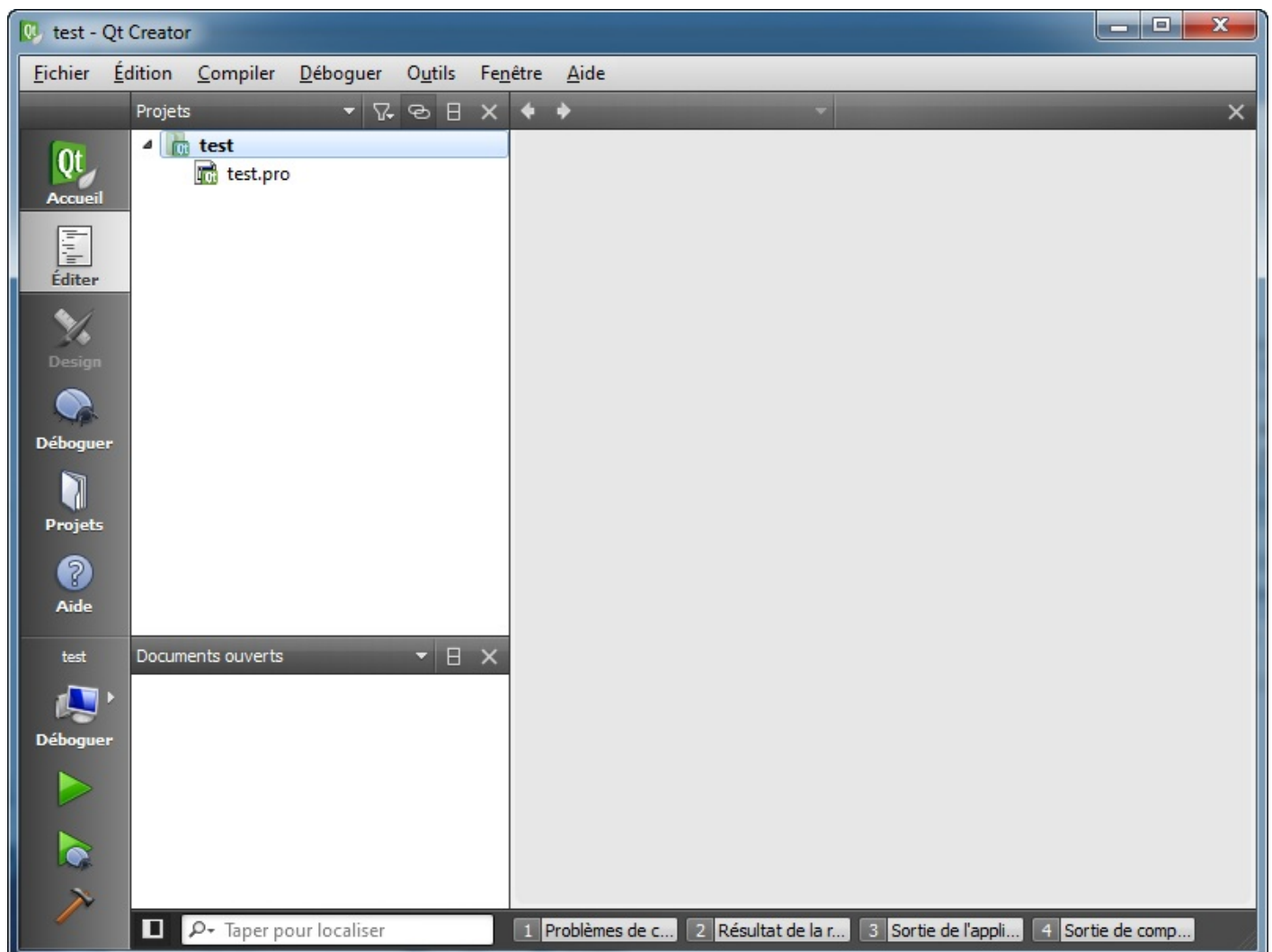
Pour le moment, faites simplement "Suivant".

Pour cette dernière fenêtre, Qt Creator propose de configurer la compilation. Là encore, laissez ce qu'on vous propose par défaut, cela conviendra très bien.



Nouveau projet (suite)

Ouf ! Notre projet vide est créé :



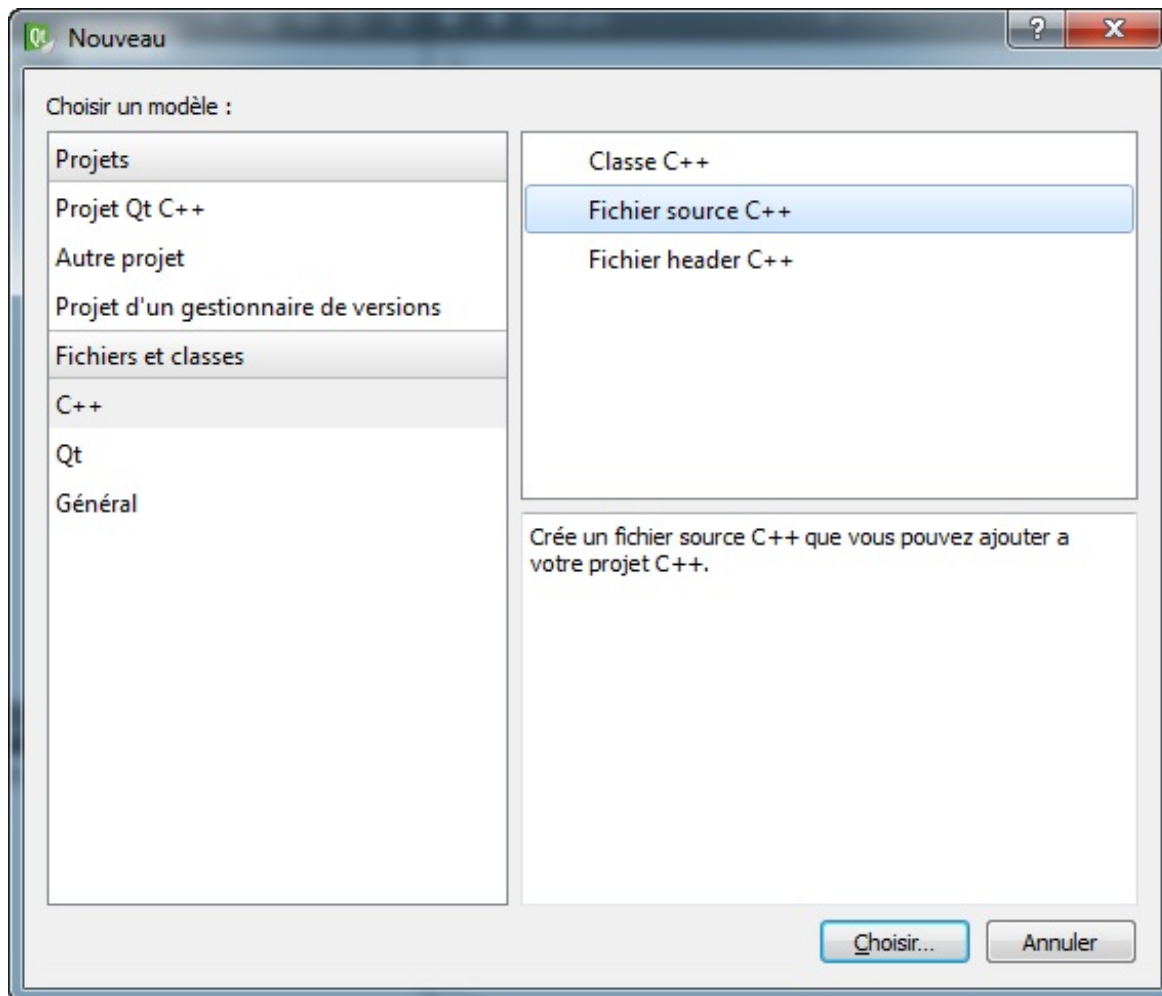
Projet vide Qt Creator

Le projet est constitué seulement d'un fichier `.pro`. Ce fichier, propre à Qt, sert à configurer le projet au moment de la compilation. Qt Creator modifie automatiquement ce fichier en fonction des besoins, ce qui fait que nous n'aurons pas beaucoup à le modifier dans la pratique.

Ajout d'un fichier `main.cpp`

Il nous faut au moins un fichier `main.cpp` pour commencer à programmer !

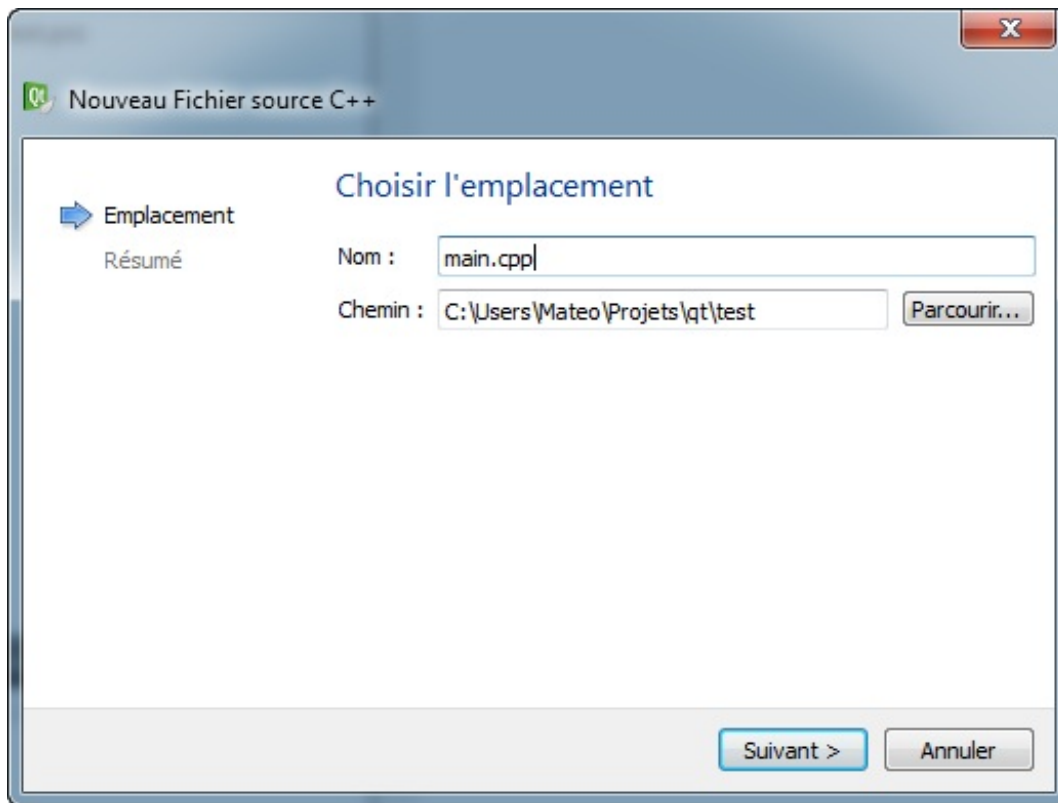
Pour l'ajouter, retournez dans le menu `Fichier / Nouveau fichier ou projet` et sélectionnez cette fois `C++ / Fichier source C++` pour ajouter un fichier `.cpp` :



Ajout de fichier

source

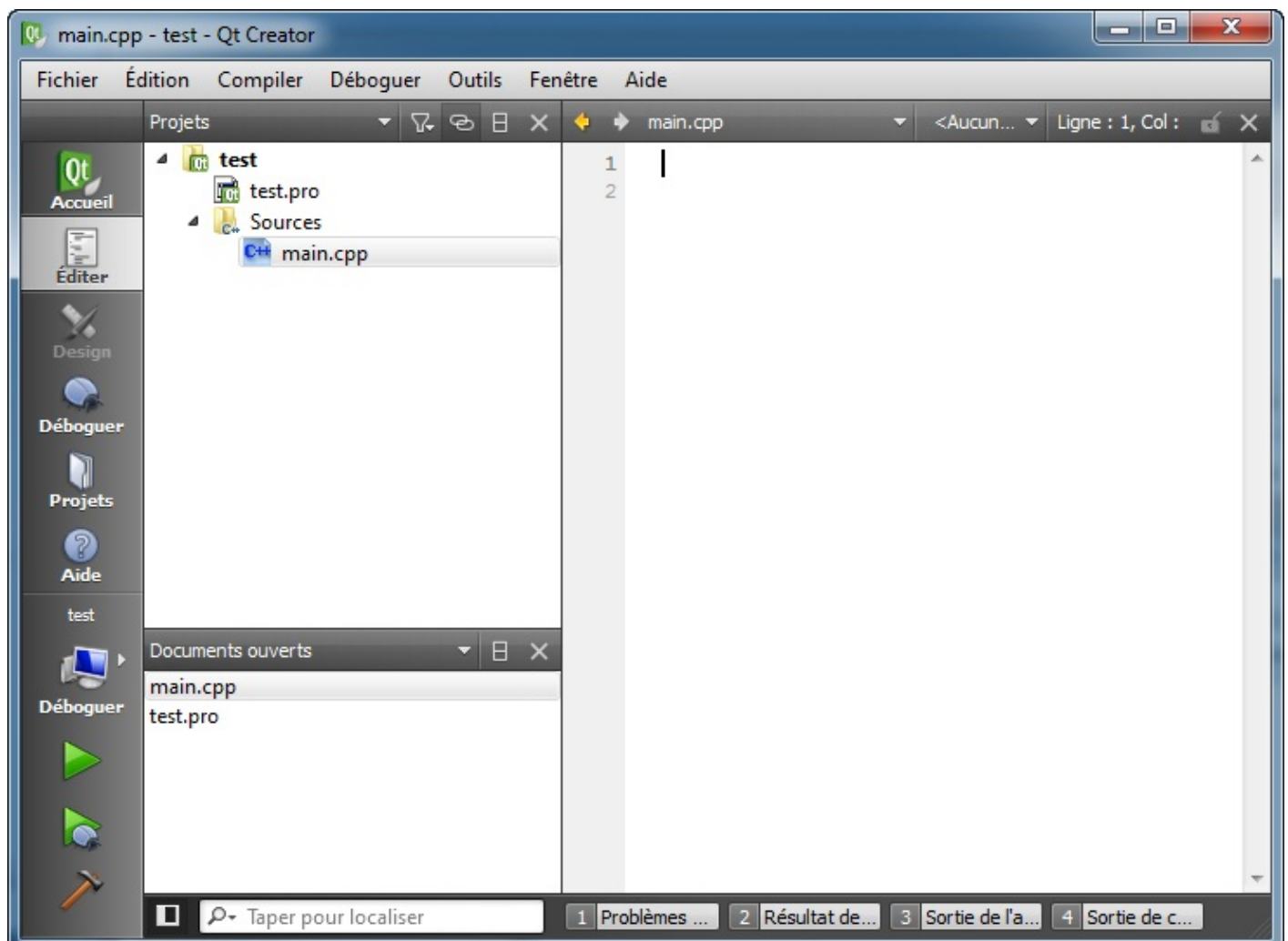
On vous demande ensuite le nom du fichier à créer, indiquez `main.cpp` :



Ajout de fichier source

(suite)

Le fichier `main.cpp` vide a été ajouté au projet. Vous pouvez double-cliquer sur son nom pour l'ouvrir :



Projet Qt vide avec main

C'est dans ce fichier que nous écrirons nos premières lignes de code C++ utilisant Qt. 😊

Pour compiler le programme, il vous suffira de cliquer sur la flèche verte dans la colonne à gauche, ou bien d'utiliser le raccourci clavier Ctrl+R.

Codons notre première fenêtre !

Ok on est parti !

Le code minimal d'un projet Qt

Rentrez le code suivant dans le fichier `main.cpp` :

Code : C++

```
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    return app.exec();
}
```

C'est le **code minimal** d'une application utilisant Qt !

Comme vous pouvez le constater, ce qui est génial c'est que c'est vraiment très court. 😊

D'autres bibliothèques vous demandent beaucoup plus de lignes de code avant de pouvoir commencer à programmer, tandis qu'avec Qt c'est vraiment très simple et rapide. 😊

Analysons ce code pas à pas !

Includes un jour, includes toujours

Code : C++

```
#include <QApplication>
```

C'est le seul include que vous avez besoin de faire au départ. Vous pouvez oublier `iostream` et compagnie, avec Qt on ne s'en sert plus.

Vous noterez qu'on ne met pas l'extension `".h"`, c'est voulu. Faites exactement comme moi.

Cet include vous permet d'accéder à la classe `QApplication`, qui est la classe de base de tout programme Qt.

QApplication, la classe de base

Code : C++

```
QApplication app(argc, argv);
```


La première ligne du *main* crée un nouvel objet de type `QApplication`. On a fait ça tout le long des derniers chapitres, vous ne devriez pas être surpris 😊

Cet objet est appelé `app` (mais vous pouvez l'appeler comme vous voulez). Le constructeur de `QApplication` exige que vous lui passiez les arguments du programme, c'est-à-dire les paramètres `argc` et `argv` que reçoit la fonction *main*. Cela permet de démarrer le programme avec certaines options précises, mais on ne s'en servira pas ici.

Lancement de l'application

Code : C++

```
return app.exec();
```

Cette ligne fait 2 choses :

1. Elle appelle la méthode `exec` de notre objet `app`. Cette méthode démarre notre programme et lance donc l'affichage des fenêtres. Si vous ne le faites pas il ne se passera rien.
2. Elle retourne le résultat de `app.exec()` pour dire si le programme s'est bien déroulé ou pas. Le `return` provoque la fin de la fonction *main*, donc du programme.

C'est un peu du condensé en fait 😊

Ce que vous devez vous dire, c'est qu'en gros tout notre programme s'exécute réellement à partir de ce moment-là. La méthode `exec` est gérée par Qt : tant qu'elle s'exécute, notre programme est ouvert. Dès que la méthode `exec` est terminée, notre programme s'arrête.

Affichage d'un widget

Dans la plupart des bibliothèques GUI, dont Qt fait partie, tous les éléments d'une fenêtre sont appelés des **widgets**. Les boutons, les cases à cocher, les images... tout ça ce sont des widgets. La fenêtre elle-même est considérée comme un widget.

Pour provoquer l'affichage d'une fenêtre, il suffit de demander à afficher n'importe quel widget. Ici par exemple, nous allons afficher un bouton.

Voici le code complet que j'aimerais que vous utilisiez. Il utilise le code de base de tout à l'heure mais y ajoute quelques lignes :

Code : C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.show();

    return app.exec();
}
```

Les lignes ajoutées ont été surlignées pour bien que vous puissiez les repérer. On voit entre autres :

Code : C++

```
#include <QPushButton>
```

Cette ligne va vous permettre de créer des objets de type QPushButton, c'est-à-dire des boutons (vous noterez que sous Qt toutes les classes commencent par un "Q" d'ailleurs !).

Code : C++

```
QPushButton bouton("Salut les Zéros, la forme ?");
```

Cela crée un nouvel objet de type QPushButton que nous appelons tout simplement *bouton*, mais on aurait très bien pu l'appeler autrement. Le constructeur attend un paramètre : le texte qui sera affiché sur le bouton.

Malheureusement, le fait de créer un bouton ne suffit pas pour qu'il soit affiché. Il faut appeler sa méthode *show* :

Code : C++

```
bouton.show();
```

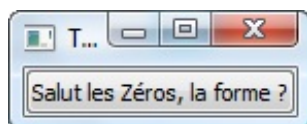
Et voilà !

Cette ligne commande l'affichage d'un bouton. Comme un bouton ne peut pas "flotter" comme ça sur votre écran, Qt l'insère automatiquement dans une fenêtre. On a en quelque sorte créé une "fenêtre-bouton" 🤪

Bien entendu, dans un vrai programme plus complexe, on crée d'abord une fenêtre et on y insère ensuite plusieurs widgets, mais là on commence simplement 🤪

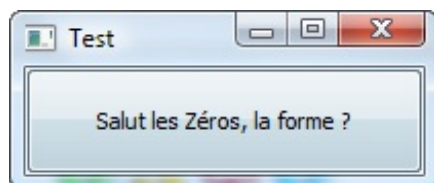
Notre code est prêt, il ne reste plus qu'à compiler et exécuter le programme ! Il suffit pour cela de cliquer sur le bouton en forme de flèche verte à gauche de Qt Creator.

Le programme se lance alors... Coucou petite fenêtre, fais risette à la caméra !



Le bouton prend la taille du texte qui se trouve à l'intérieur, et la fenêtre qui est automatiquement créée prend la taille du bouton. Ça donne donc une toute petite fenêtre 🤪

Mais... vous pouvez la redimensionner, voire même l'afficher en plein écran ! Rien ne vous en empêche, et le bouton s'adapte automatiquement à la taille de la fenêtre (ce qui peut donner un très très gros bouton) :



Un peu de théorie : Qt et la compilation

La bibliothèque Qt est tellement importante qu'elle apporte quelques ajouts au langage C++, en particulier le mécanisme des signaux et slots dont on reparlera un peu plus loin.

Pour ajouter ces fonctionnalités au langage C++, la compilation sort du schéma classique. On va en profiter ici pour revoir (ou voir pour certains 🤪) comment la compilation fonctionne d'habitude dans les grandes lignes, puis nous découvrirons comment sont compilés les programmes Qt.



Vous n'êtes pas obligés de connaître tout cela car Qt Creator effectue tout le travail pour nous, mais il peut être intéressant pour votre culture de savoir comment la compilation fonctionne en coulisses. 😊

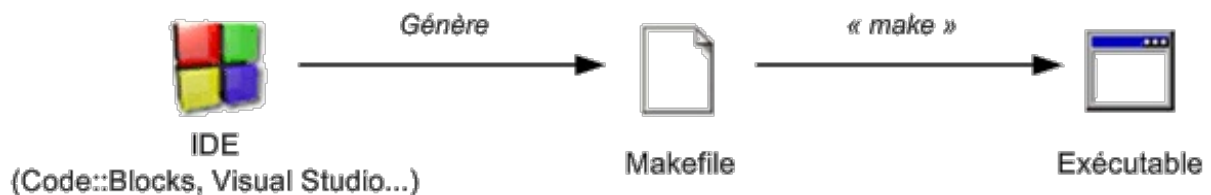
La compilation "normale", sans Qt

Savez-vous vraiment ce qui se passe lorsque vous cliquez sur "Compiler" dans votre IDE habituel (Code::Blocks, Visual Studio...)?

Il y a en fait 2 grosses étapes :

1. L'IDE regarde la liste des fichiers de votre projet (.cpp et .h) et génère un fichier appelé Makefile qui contient la liste des fichiers à compiler pour le compilateur.
2. Ensuite, l'IDE appelle le compilateur (via le programme *make*). L'utilitaire *make* recherche un fichier Makefile et l'utilise pour savoir quoi compiler et avec quelles options.

Schématiquement ça donne ça :



En fait, le gros avantage de l'IDE c'est qu'il écrit le fichier Makefile pour vous. On peut écrire le Makefile à la main, mais c'est honnêtement pas très pratique ni toujours très simple, surtout pour de gros projets. En effet, le fichier Makefile est parfois très gros.

Voici un aperçu (raccourci) d'un fichier Makefile pour vous donner une idée :

Code : Autre

```

all: $(PROG)

$(PROG): $(OBJS)
    $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)

.c.o:
    $(CC) $(CFLAGS) -c $*.c
  
```

Qu'un IDE génère le Makefile pour nous n'est donc pas du luxe 😊

La compilation particulière avec Qt

Le problème survient quand vous utilisez des bibliothèques importantes comme Qt. Il faut configurer votre IDE pour qu'il puisse écrire le Makefile correctement, c'est-à-dire indiquer l'emplacement des fichiers .a (ou .lib), des headers de la bibliothèque, etc.

On pourrait en théorie configurer votre IDE pour que ça marche en cliquant sur "Compiler"... mais ce serait un peu long et compliqué (il faudrait une explication par IDE, et parfois par version d'IDE). J'ai à la place choisi de vous montrer une technique universelle : passer par la ligne de commande ! 😊



Bah pourquoi tout le monde est parti ?

Je vous rassure, ce n'est pas aussi infaisable que ce que vous pouvez croire, ce sera même plus simple que de configurer notre IDE, c'est vous dire 😊

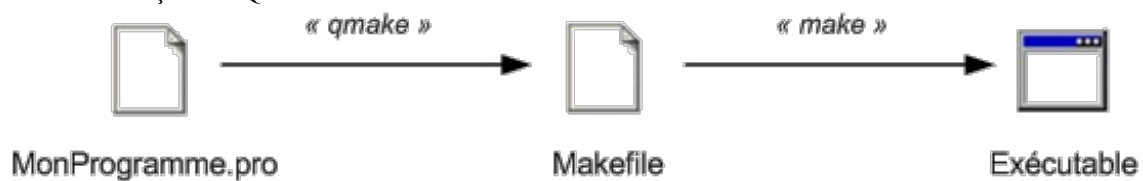
Qt est livré avec un petit programme en ligne de commande appelé "*qmake*". Ce programme est capable de générer un fichier Makefile à partir d'un fichier spécifique à Qt : le .pro.

Le .pro (qui s'appelle en général nomDeVôtreProjet.pro) est un fichier texte court et simple à écrire qui donne la liste de vos fichiers .cpp et .h, ainsi que les options à envoyer à Qt.



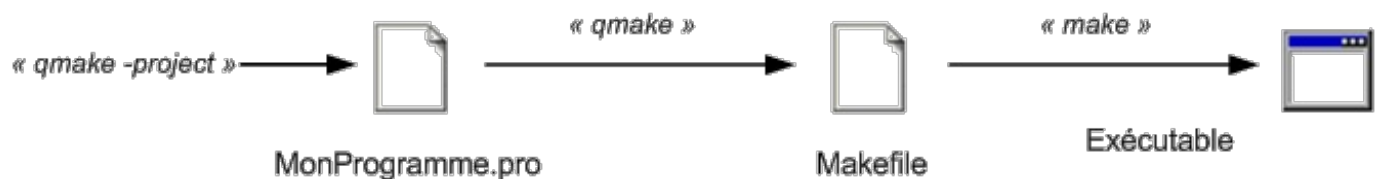
Sous Linux, la commande n'est pas *qmake* mais *qmake-qt4*.

Ca se passe donc comme ça avec Qt :



... Je suis obligé d'écrire moi-même le .pro ? Je ne sais pas faire ! Et puis faire la liste des fichiers du projet ça peut être long non ? 😊

Rassurez-vous, Qt peut vous générer un .pro automatiquement ! Si on utilise d'abord *qmake* avec l'option *-project* dans le dossier de notre projet, *qmake* va analyser les fichiers du dossier et générer un fichier .pro basique (mais suffisant pour nous pour le moment).



En résumé, pour compiler avec Qt il y a 3 commandes très simples à taper en console. Dans l'ordre :

1. *qmake -project*
2. *qmake*
3. *make* (sous Linux) ou *mingw32-make* (sous Windows)

Normalement, il n'est nécessaire de taper les 2 premières commandes (*qmake -project* et *qmake*) que la première fois pour générer le Makefile. Ensuite, vous n'aurez plus besoin que de relancer *make* pour recompiler votre projet.



Il faudra en fait relancer les commandes *qmake -project* et *qmake* à chaque fois que votre projet évoluera, c'est-à-dire à chaque fois que de nouveaux fichiers .cpp et .h seront ajoutés ou supprimés. Tant que la liste des fichiers de votre projet ne change pas, il n'est pas nécessaire de retaper ces 2 premières commandes.

Bien entendu, si vous utilisez Qt Creator, tout cela est fait automatiquement pour vous de façon transparente, vous n'avez pas à vous en préoccuper ! Cependant, connaître le fonctionnement de la compilation ne peut être qu'un "plus" pour vous. 😊

Diffuser le programme

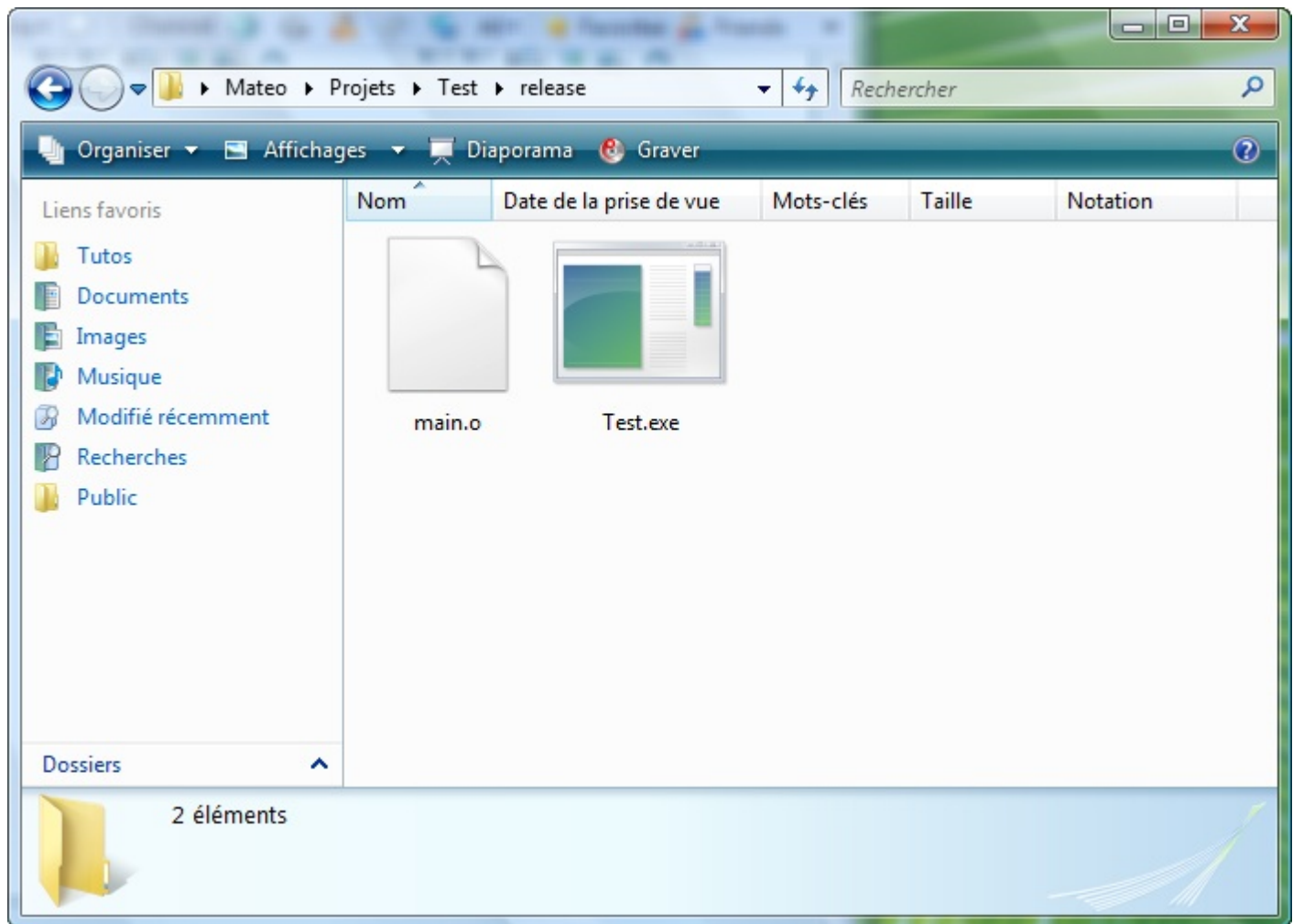
Pour tester le programme avec Qt Creator, un clic sur la flèche verte suffit. C'est très simple.

Cependant, si vous récupérez l'exécutable qui a été généré et que vous l'envoyez à un ami, celui-ci ne pourra probablement pas lancer votre programme ! En effet, il a besoin d'une série de fichiers DLL.

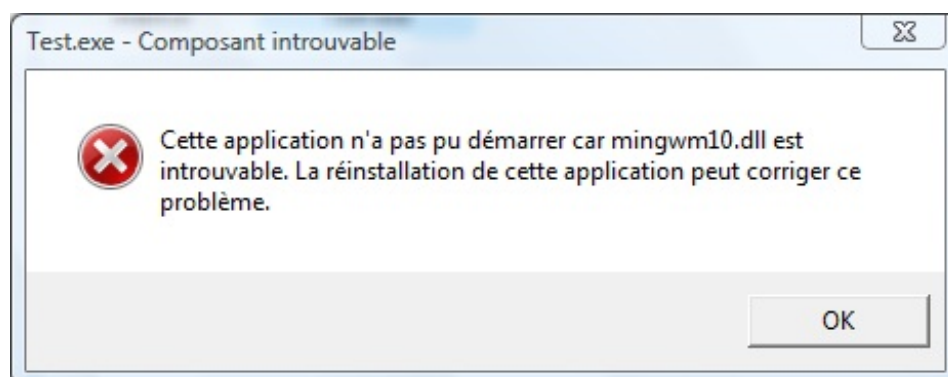
Les programmes Qt ont besoin de ces fichiers DLL avec eux pour fonctionner.

Quand vous exécutez votre programme depuis Qt Creator, la position des DLL est "connue", donc votre programme se lance sans erreur.

Mais essayez de double-cliquer sur l'exécutable depuis l'explorateur pour voir ! Rendez-vous dans le sous-dossier "release" de votre projet pour y trouver l'exécutable :



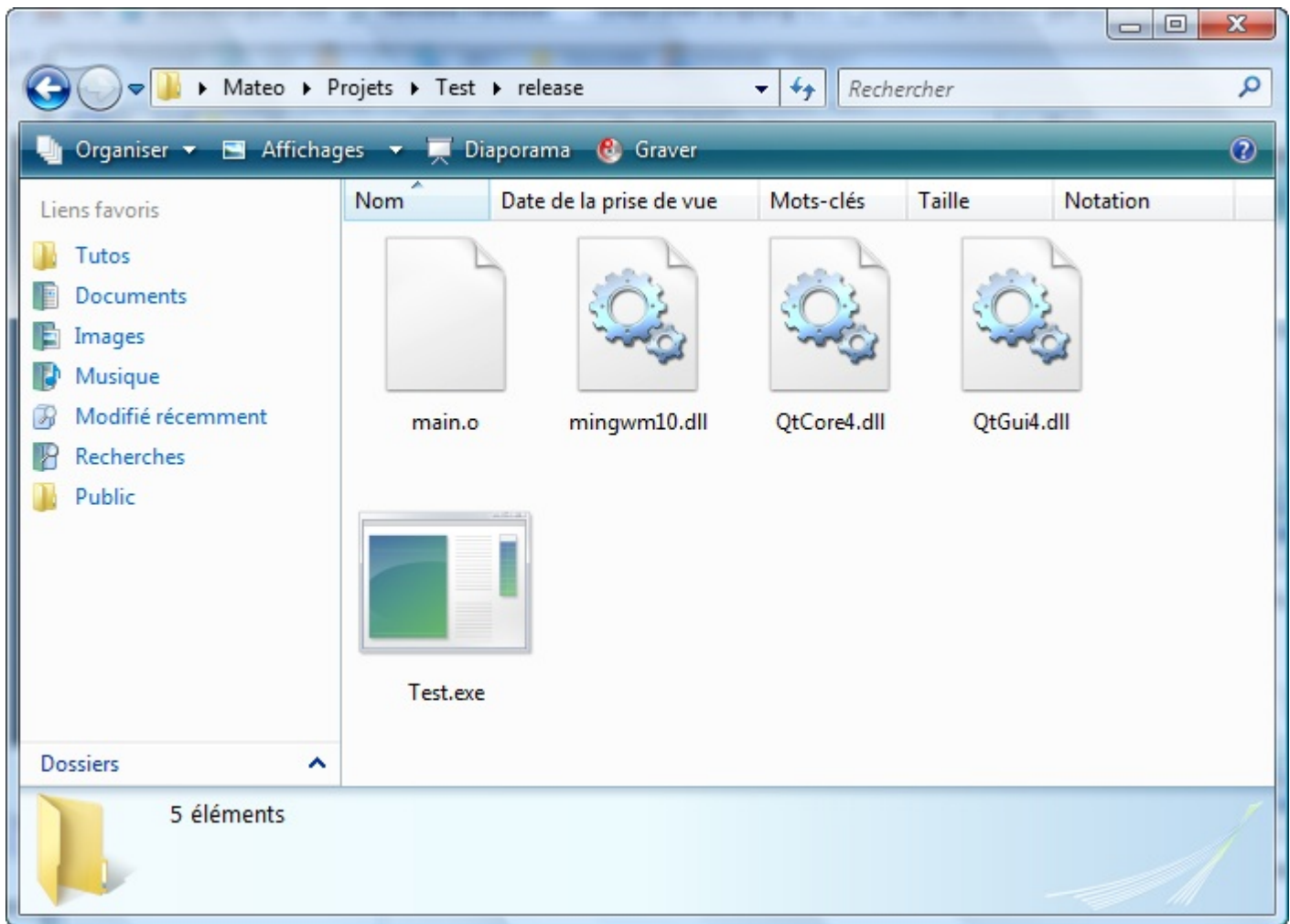
Le programme Test.exe. Double-cliquez dessus.



Miséricorde ! Ca ne marche pas !

En effet, sans quelques DLL à côté notre programme est perdu. Il a besoin de ces fichiers qui contiennent de quoi le guider.

Pour pouvoir lancer l'exécutable depuis l'explorateur (et aussi pour qu'il marche chez vos amis / clients), il faut placer les DLL qui manquent dans le même dossier que l'exécutable. A vous de les chercher, vous les avez sur votre disque (chez moi je les ai trouvés dans le dossier C:\Qt\2010.05\mingw\bin et C:\Qt\2010.05\bin). En tout, vous devriez avoir eu besoin de mettre 3 DLL :



Vous pouvez lancer le programme depuis l'explorateur maintenant !

Lorsque vous envoyez votre programme à un ami ou que vous le mettez en ligne pour téléchargement, pensez donc à joindre les DLL, elles sont indispensables.

Nous y sommes enfin arrivés, champagne ! 🍾

Vous l'avez vu, le code nécessaire pour ouvrir une fenêtre toute simple constituée d'un bouton est ridicule. Quelques lignes à peine, et rien de bien compliqué à comprendre au final.

C'est ce qui fait la force de Qt : *"un code simple est un beau code"* dit-on. Qt s'efforce de respecter ce dicton à la lettre, vous vous en rendrez compte dans les prochains chapitres.

Dans les prochains chapitres, nous allons voir comment changer l'apparence du bouton, comment faire une fenêtre un peu plus complexe. Nous découvrirons aussi le mécanisme des signaux et des slots, un des principes les plus importants de Qt qui permet de gérer les événements : un clic sur un bouton pourra par exemple provoquer l'ouverture d'une nouvelle fenêtre ou sa fermeture !

Personnaliser les widgets

La "fenêtre-bouton" que nous avons réalisée dans le chapitre précédent était un premier pas. Toutefois, nous avons passé plus de temps à expliquer les mécanismes de la compilation qu'à modifier le contenu de la fenêtre.

Par exemple, comment faire pour modifier la taille du bouton ? Comment placer le bouton où on veut sur la fenêtre ? Comment modifier les propriétés du bouton ? Changer la couleur, le curseur de la souris, la police, l'icône...

Dans ce chapitre, nous allons nous habituer à modifier les propriétés d'un widget : le bouton. Bien sûr, il existe des tonnes d'autres widgets (cases à cocher, listes déroulantes...) mais nous nous concentrerons sur le bouton pour nous habituer à éditer les propriétés d'un widget.

Une fois que vous saurez le faire pour le bouton, vous n'aurez aucun mal à le faire pour les autres widgets.

Enfin et surtout, nous reparlerons d'héritage dans ce chapitre. Nous apprendrons à créer un widget personnalisé qui "hérite" du bouton. C'est une technique extrêmement courante que l'on retrouve dans toutes les bibliothèques de création de GUI !

Allez hop, vous allez me personnaliser ce bouton tout gris !

"Yo man, on va te cus-to-mi-ser ton vieux bouton à la sauce west coast ! Aujourd'hui sur le Site du Zéro, c'est Pimp mon bouton !"

Pardonnez ce petit délire, je promets à l'avenir de ne plus regarder MTV avant de rédiger un tutoriel. Promis promis.



Modifier les propriétés d'un widget

Comme tous les éléments d'une fenêtre, on dit que le bouton est un widget. Avec Qt, on crée un bouton à l'aide de la classe QPushButton.

Comme vous le savez, une classe est constituée de 2 éléments :

- **Des attributs** : ce sont les "variables" internes de la classe.
- **Des méthodes** : ce sont les "fonctions" internes de la classe.

La règle d'encapsulation dit que les utilisateurs de la classe ne doivent pas pouvoir modifier les attributs : ceux-ci doivent donc tous être privés.

Or, je ne sais pas si vous avez remarqué, mais nous sommes justement des *utilisateurs* des classes de Qt. Ce qui veut dire... que nous n'avons pas accès aux attributs puisque ceux-ci sont privés ! 🤔



Hé, mais tu avais parlé d'un truc à un moment je crois... Les accesseurs, c'est pas ça ?

Ah... J'aime les gens qui ont de la mémoire 😊

Effectivement oui, j'avais dit que le créateur d'une classe devait rendre ses attributs privés, mais du coup proposer des méthodes *accesseurs*, c'est-à-dire des méthodes permettant de lire et de modifier les attributs de manière sécurisée (get et set ça vous dit rien ?).

Les accesseurs avec Qt

Justement, les gens qui ont créé Qt chez Trolltech sont des braves gars : ils ont codé proprement en respectant ces règles. Et il valait mieux qu'ils fassent bien les choses s'ils ne voulaient pas que leur bibliothèque devienne un véritable foutoir !

Du coup, pour chaque propriété d'un widget, on a :

- **Un attribut** : il est privé on ne peut pas le lire ni le modifier directement.
Exemple : text
- **Un accesseur pour le lire** : cet accesseur est une méthode constante qui porte le même nom que l'attribut (personnellement j'aurais plutôt mis un "get" devant pour ne pas confondre avec l'attribut, mais bon). Je vous rappelle qu'une méthode constante est une méthode qui s'interdit de modifier les attributs de la classe. Ainsi, vous êtes assuré que la méthode ne fait que lire l'attribut et qu'elle ne le modifie pas.

Exemple : `text()`

- **Un accesseur pour le modifier** : c'est une méthode qui se présente sous la forme `setAttribut()`. Elle modifie la valeur de l'attribut.

Exemple : `setText()`

Cette technique, même si elle paraît un peu lourde parce qu'il faut créer 2 méthodes pour chaque attribut, a l'avantage d'être parfaitement sûre. Grâce à ça, Qt peut vérifier que la valeur que vous essayez de donner est valide.

Cela permet d'éviter par exemple que vous ne donniez à une barre de progression la valeur "150%", alors que la valeur d'une barre de progression doit être comprise entre 0 et 100%.



Voyons voir sans plus tarder quelques propriétés des boutons que nous pouvons nous amuser à modifier à l'aide des accesseurs



Quelques exemples de propriétés des boutons

Il existe un grand nombre de propriétés éditables pour chaque widget, y compris le bouton. Nous n'allons pas toutes les voir ici, ni même plus tard d'ailleurs, je vous apprendrai à lire la doc pour toutes les découvrir



Cependant, je tiens à vous montrer les plus intéressantes d'entre elles pour que vous puissiez commencer à vous faire la main, et surtout pour que vous preniez l'habitude d'utiliser les accesseurs de Qt.

text : le texte

Cette propriété est probablement la plus importante : elle permet de modifier le texte présent sur le bouton.

En général, on définit le texte du bouton au moment de sa création car le constructeur accepte que l'on donne le texte du bouton dès sa création.

Toutefois, pour une raison ou une autre, vous pourriez être amené à modifier le texte présent sur le bouton au cours de l'exécution du programme. C'est là qu'il devient pratique d'avoir accès à l'attribut "text" du bouton *via* ses accesseurs.

Pour chaque attribut, la documentation de Qt nous dit à quoi il sert et quels sont ses accesseurs. Voyez par exemple [ce que ça donne pour l'attribut text des boutons](#).

On vous indique de quel type est l'attribut. Ici, text est de type `QString`, comme tous les attributs qui stockent du texte avec Qt. En effet, Qt n'utilise pas la classe "string" standard du C++ mais sa propre version de la gestion des chaînes de caractères. En gros, *QString* c'est un *string* amélioré.

Puis, on vous explique en quelques mots à quoi sert cet attribut (*in english of course*, il n'est jamais trop tard pour reprendre des cours d'anglais quel que soit votre âge



Enfin, on vous indique les accesseurs qui permettent de lire et de modifier l'attribut. Dans le cas présent, il s'agit de :

- `QString text () const` : c'est l'accesseur qui permet de **lire l'attribut**. Il retourne un `QString`, ce qui est logique puisque l'attribut est de type `QString`. Vous noterez la présence du mot-clé "const" qui indique que c'est une méthode constante qui ne modifie aucun attribut.
- `void setText (const QString & text)` : c'est l'accesseur qui permet de **modifier l'attribut**. Il prend un paramètre : le texte que vous voulez mettre sur le bouton.



A la longue, vous ne devriez pas avoir besoin de la doc pour savoir quels sont les accesseurs d'un attribut. Ça suit toujours le même schéma :

`attribut()` : permet de lire l'attribut.

`setAttribut()` : permet de modifier l'attribut.

Essayons donc de modifier le texte du bouton après sa création :

Code : C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.setText("Pimp mon bouton !");

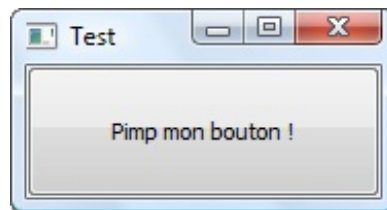
    bouton.show();

    return app.exec();
}
```



Vous aurez noté que la méthode `setText` attend un `QString` et qu'on lui envoie une bête chaîne de caractères entre guillemets. En fait, ça fonctionne comme la classe `string` : les chaînes de caractères entre guillemets sont automatiquement converties en `QString`. Heureusement d'ailleurs, sinon ça serait lourd de devoir créer un objet de type `QString` juste pour ça !

Résultat :



Le résultat n'est peut-être pas très impressionnant, mais ça montre bien ce qui se passe :

1. On crée le bouton et on lui donne le texte "Salut les Zéros, la forme ?" à l'aide du constructeur.
2. On modifie le texte présent sur le bouton pour afficher "Pimp mon bouton !".

Au final, c'est "Pimp mon bouton !" qui s'affiche.

Pourquoi ? Parce que le nouveau texte a "écrasé" l'ancien. C'est exactement comme si on faisait :

Code : C++

```
int x = 1;
x = 2;
cout << x;
```

... Lorsqu'on affiche `x`, il vaut 2.

C'est pareil pour le bouton. Au final, c'est le tout dernier texte qui sera affiché.

Bien entendu, ce qu'on vient de faire est complètement inutile : autant donner le bon texte directement au bouton lors de l'appel du constructeur. Toutefois, `setText()` se révèlera utile plus tard lorsque vous voudrez modifier le contenu du bouton au cours de l'exécution. Par exemple, lorsque l'utilisateur aura donné son nom, le bouton pourra changer de texte pour dire "Bonjour M. Dupont !".

toolTip : l'infobulle

Il est courant d'afficher une petite aide sous la forme d'une infobulle qui apparaît lorsqu'on pointe sur un élément avec la souris.

L'infobulle peut afficher un court texte d'aide. On la définit à l'aide de la propriété `toolTip`.

Pour modifier l'infobulle, la méthode à appeler est donc... `setToolTip` ! Bah vous voyez, c'est facile quand on a compris comment Qt était organisé 😊

Code : C++

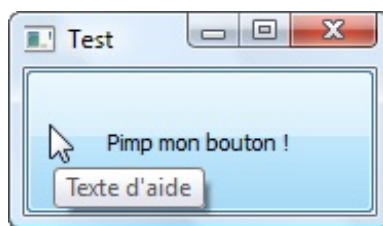
```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Pimp mon bouton !");
    bouton.setToolTip("Texte d'aide");

    bouton.show();

    return app.exec();
}
```



Une infobulle

font : la police

Avec la propriété `font`, les choses se compliquent. En effet, jusqu'ici on avait juste eu à envoyer une chaîne de caractères en paramètres, qui était en fait convertie en objet de type `QString`.

La propriété `font` est un peu plus complexe car elle contient 3 informations :

- Le nom de la police de caractères utilisée (Times New Roman, Arial, Comic Sans MS...)
- La taille du texte en pixels (12, 16, 18...)
- Le style du texte (gras, italique...)

La signature de la méthode `setFont` est :

```
void setFont ( const QFont & )
```

Cela veut dire que `setFont` attend un objet de type `QFont` !

Je rappelle, pour ceux qui auraient oublié la signification des symboles, que :



- **const** : signifie que l'objet que l'on envoie en paramètre ne sera pas modifié par la fonction
- **&** : signifie que la fonction attend une référence vers l'objet. En C, il aurait fallu envoyer un pointeur, mais comme en C++ on dispose des références (qui sont plus simples à utiliser), on en profite 😊

Bon, comment on fait pour lui donner un objet de type `QFont` nous ?

Eh bien c'est simple : il... suffit de créer un objet de type `QFont` !

La doc nous indique [tout ce que nous avons besoin de savoir sur QFont](#), en particulier les informations qu'il faut donner à son constructeur. Je n'attends pas de vous encore que vous soyez capable de lire la doc de manière autonome, je vais donc vous mâcher le travail (mais profitez-en parce que ça ne durera pas éternellement 🤖).

Pour faire simple, le constructeur de QFont attend 4 paramètres. Voici son prototype :

`QFont (const QString & family, int pointSize = -1, int weight = -1, bool italic = false)`



En fait, avec Qt il y a rarement un seul constructeur par classe. Les développeurs de Qt profitent des fonctionnalités du C++ et ont donc tendance à beaucoup surcharger les constructeurs. Certaines classes possèdent même plusieurs dizaines de constructeurs différents ! Pour QFont, celui que je vous montre là est néanmoins le principal et le plus utilisé. Et le plus simple aussi, tant qu'à faire.

Seul le premier argument est obligatoire : il s'agit du nom de la police à utiliser. Les autres, comme vous pouvez le voir, possèdent des valeurs par défaut donc nous ne sommes pas obligés de les indiquer.

Dans l'ordre, les paramètres signifient :

- family : le nom de la police de caractères à utiliser.
- pointSize : la taille des caractères en pixels.
- weight : le niveau d'épaisseur du trait (gras). Cette valeur peut être comprise entre 0 et 99 (du plus fin au plus gras). Vous pouvez aussi utiliser la constante `QFont::Bold` qui correspond à une épaisseur de 75.
- italic : un booléen pour dire si le texte doit être affiché en italique ou non.

On va faire quelques tests. Tout d'abord, il va falloir créer un objet de type QFont :

Code : C++

```
QFont maPolice("Courier");
```

J'ai appelé cet objet `maPolice`.

Maintenant, je dois envoyer l'objet `maPolice` de type QFont à la méthode `setFont` de mon bouton (suivez, suivez !) :

Code : C++

```
bouton.setFont(maPolice);
```

En résumé, j'ai donc dû écrire 2 lignes pour changer la police :

Code : C++

```
QFont maPolice("Courier");  
bouton.setFont(maPolice);
```

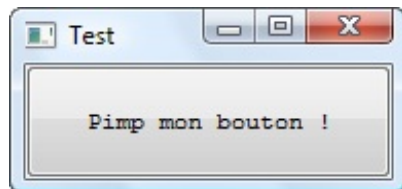
C'est un peu fastidieux. Il existe une solution plus maligne, si on ne compte pas se resservir de la police plus tard, c'est de définir l'objet de type QFont au moment de l'appel à la méthode `setFont`. Ça nous évite d'avoir à donner un nom bidon à l'objet comme on l'a fait ici (`maPolice`), c'est plus court, ça va plus vite, bref c'est mieux en général 🤖

Code : C++

```
bouton.setFont(QFont("Courier"));
```

Voilà, en imbriquant comme ça ça marche très bien. La méthode `setFont` veut un objet de type `QFont` ? Qu'à cela ne tienne, on lui en crée un à la volée !

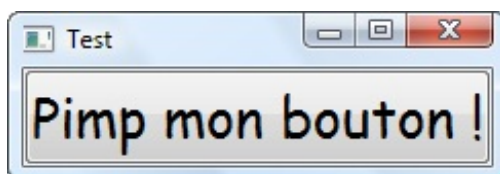
Voici le résultat :



Maintenant, on peut exploiter un peu plus le constructeur de `QFont` en utilisant une autre police plus fantaisiste et en augmentant la taille des caractères :

Code : C++

```
bouton.setFont(QFont("Comic Sans MS", 20));
```



Et voilà le même avec du gras et de l'italique !

Code : C++

```
bouton.setFont(QFont("Comic Sans MS", 20, QFont::Bold, true));
```



Bref, si vous avez compris le [principe des paramètres par défaut](#) (et j'espère que vous avez compris depuis le temps ! 🤖), ça ne devrait vous poser aucun problème.

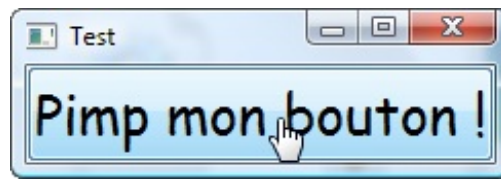
cursor : le curseur de la souris

Avec la propriété `cursor`, vous pouvez déterminer quel curseur de la souris doit s'afficher lorsqu'on pointe sur le bouton. Le plus simple est d'utiliser une des [constantes de curseurs prédéfinis](#) parmi la liste qui s'offre à vous.

Ce qui peut donner par exemple, si on veut qu'une main s'affiche :

Code : C++

```
bouton.setCursor(Qt::PointingHandCursor);
```



icon : l'icône du bouton

Après tout ce qu'on vient de voir, rajouter une icône au bouton va vous paraître très simple : la méthode `setIcon` attend juste un objet de type `QIcon`.

Un `QIcon` peut se construire très facilement en donnant le nom du fichier image à charger.

Prenons par exemple ce petit smiley souriant : 😊

Il s'agit d'une image au format PNG que sait lire Qt.

Code : C++

```
bouton.setIcon(QIcon("smile.png"));
```

Attention, sous Windows pour que cela fonctionne, votre icône `smile.png` doit se trouver dans le même dossier que l'exécutable (ou dans un sous-dossier si vous écrivez `"dossier/smile.png"`).

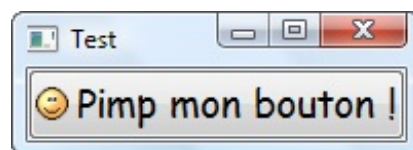
Sous Linux, il faut que votre icône soit dans votre répertoire HOME. Si vous voulez utiliser le chemin de votre application, comme cela se fait sous Windows par défaut, écrivez :

Code : C++

```
QIcon(QCoreApplication::applicationDirPath() + "/smile.png");
```

Cela aura pour effet d'afficher l'icône à condition que celle-ci se trouve dans le même répertoire que l'exécutable.

Si vous avez fait ce qu'il fallait, l'icône devrait alors apparaître comme ceci :



Qt et l'héritage

On aurait pu continuer à faire joujou longtemps avec les propriétés de notre bouton, mais il faut savoir s'arrêter au bout d'un moment et reprendre les choses sérieuses.

Quelles choses sérieuses ?

Si je vous dis "héritage", ça ne vous rappelle rien ? J'espère que ça ne vous donne pas des boutons en tout cas (oh oh oh), parce que si vous n'avez pas compris le [principe de l'héritage](#) vous ne pourrez pas aller plus loin.

De l'héritage en folie

L'héritage est probablement LA notion la plus intéressante de la programmation orientée objet. Le fait de pouvoir créer une classe de base, réutilisée par des sous-classes filles, qui ont elles-mêmes leurs propres sous-classes filles, ça donne à une bibliothèque comme Qt une puissance infinie (voire plus, même).

En fait... quasiment toutes les classes de Qt font appel à l'héritage.

Pour vous faire une idée, la documentation vous donne la [hiérarchie complète des classes](#). Chaque classe "à gauche" de cette liste à puces est une classe de base, et les classes qui sont décalées vers la droite sont des sous-classes.

Vous pouvez par exemple voir au début :

- QAbstractExtensionFactory
 - QExtensionFactory
- QAbstractExtensionManager
 - QExtensionManager

QAbstractExtensionFactory et QAbstractExtensionManager sont des classes dites "de base". Elles n'ont pas de classes parentes.

En revanche, QExtensionFactory et QExtensionManager sont des classes-filles, qui héritent respectivement de QAbstractExtensionFactory et QAbstractExtensionManager.

Sympa hein ? 😊

Descendez plus bas sur la [page de la hiérarchie](#) à la recherche de la classe QObject.

Regardez un peu toutes ses classes filles.

Descendez.

Encore.

Encore.

Encore.

C'est bon vous avez pas trop pris peur ? 😊

Vous avez dû voir que certaines classes étaient carrément des sous-sous-sous-sous-sous-classes.



Wouaw mais comment je vais m'y retrouver là-dedans moi ? C'est pas possible je vais jamais m'en sortir !

C'est ce qu'on a tendance à se dire la première fois. En fait, vous allez petit à petit comprendre qu'au contraire tous ces héritages sont là pour vous simplifier la vie. Si ce n'était pas aussi bien architecturé, alors *là* vous ne vous en seriez jamais sortis !

QObject : une classe de base incontournable

QObject est la classe de base de tous les objets sous Qt.

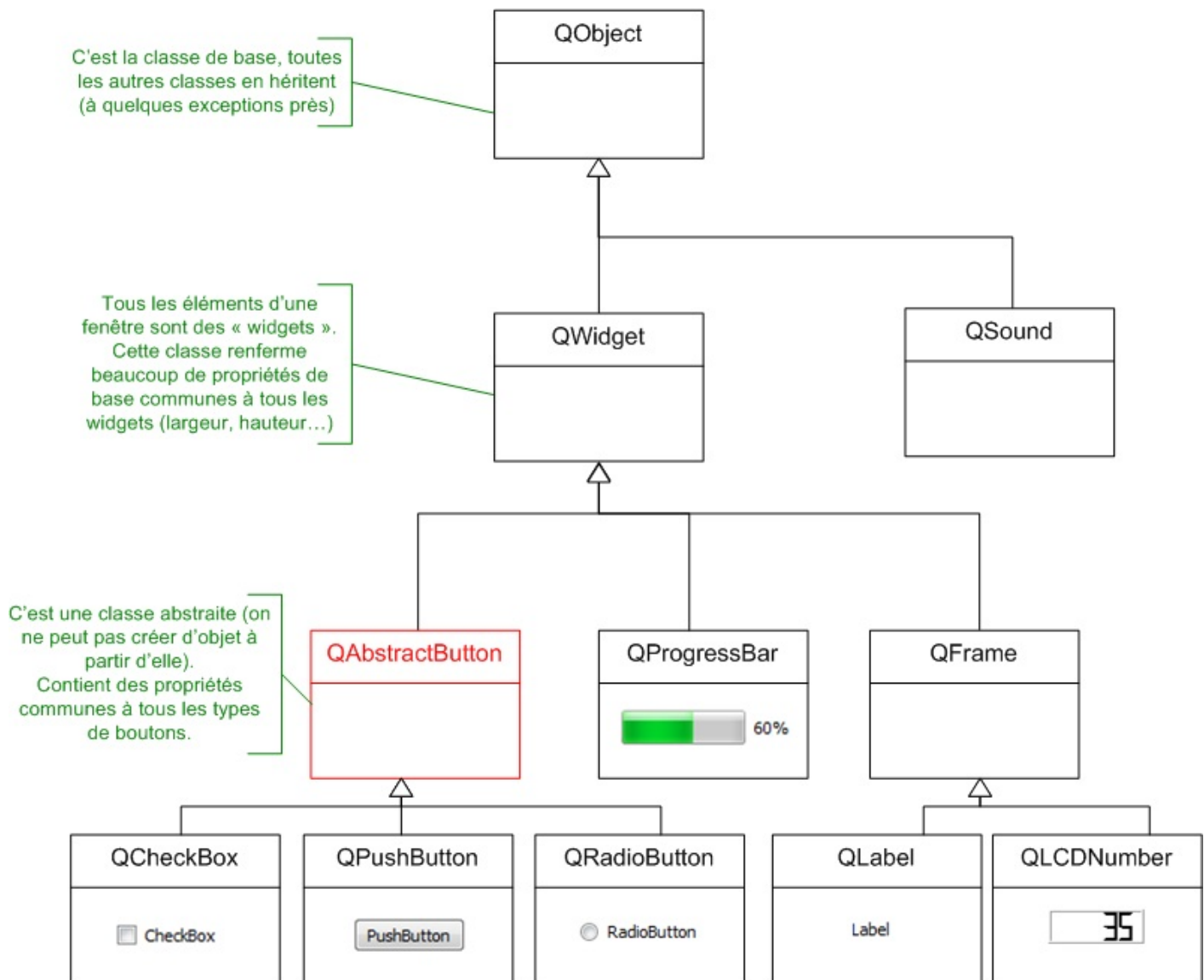
QObject ne correspond à rien de particulier, mais elle propose quelques fonctionnalités "de base" qui peuvent être utiles à toutes les autres classes.

Cela peut surprendre d'avoir une classe de base qui ne sait rien faire de particulier, mais en fait c'est ce qui donne beaucoup de puissance à la bibliothèque. Par exemple, il suffit de définir une fois dans QObject une méthode `objectName()` qui contient le nom de l'objet, et ainsi toutes les autres classes de Qt en héritent et posséderont donc cette méthode.

D'autre part, le fait d'avoir une classe de base comme QObject est indispensable pour réaliser le mécanisme des **signaux et des slots** qu'on verra dans le prochain chapitre. Ce mécanisme permet de faire en sorte par exemple que si un bouton est cliqué, alors une autre fenêtre s'ouvre (on dit qu'il envoie un signal à un autre objet).

Bref, tout cela doit vous sembler encore un peu abstrait et je le comprends parfaitement.

Je pense qu'un petit schéma simplifié des héritages de Qt s'impose. Cela devrait vous permettre de mieux visualiser la hiérarchie des classes :



Soyons clairs : je n'ai pas tout mis. J'ai juste mis quelques exemples, mais s'il fallait faire le schéma complet ça prendrait une place énorme vous vous en doutez !

On voit sur ce schéma que QObject est la classe mère principale, dont héritent toutes les autres classes. Comme je l'ai dit, elle propose quelques fonctionnalités qui se révèlent utiles pour toutes les classes, mais nous ne les verrons pas ici.

Certaines classes comme QSound (gestion du son) héritent directement de QObject.

Toutefois, comme je l'ai dit on s'intéresse plus particulièrement à la création de GUI, c'est-à-dire de fenêtres. Or, dans une fenêtre tout est considéré comme un widget (même la fenêtre est un widget).

C'est pour cela qu'il existe une classe de base QWidget pour tous les widgets. Elle contient énormément de propriétés communes à tous les widgets, comme :

- La largeur
- La hauteur
- La position en abscisse (x)
- La position en ordonnée (y)
- La police de caractères utilisée (eh oui, la méthode setFont est définie dans QWidget, et comme QPushButton en hérite, il possède lui aussi cette méthode)
- Le curseur de la souris (pareil, rebelotte, setCursor est en fait défini dans QWidget et non dans QPushButton, car il est aussi susceptible de servir sur tous les autres widgets)
- L'infobulle (toolTip)
- etc.

Vous commencez à percevoir un peu l'intérêt de l'héritage ?

Grâce à cette technique, il leur a suffi de définir **une fois** toutes les propriétés de base des widgets (largeur, hauteur...). Tous les widgets héritent de QWidget, donc ils possèdent tous ces propriétés. Vous savez donc par exemple que vous pouvez retrouver la méthode setCursor dans la classe QProgressBar.

Les classes abstraites

Vous avez pu remarquer sur mon schéma que j'ai écrit la classe QAbstractButton en rouge... Pourquoi ?

Il existe en fait un grand nombre de classes abstraites sous Qt, qui contiennent toutes le mot "Abstract" dans leur nom. Nous avons déjà parlé des [classes abstraites](#) dans un chapitre précédent.

Petit rappel pour ceux qui auraient oublié. Les classes dites "abstraites" sont des classes qu'on ne peut pas instancier. C'est-à-dire... qu'on n'a pas le droit de créer d'objet à partir d'elles. Ainsi, on ne peut pas faire :

Code : C++

```
QAbstractButton bouton() ; // Interdit car classe abstraite
```



Mais alors... à quoi ça sert de faire une classe si on ne peut pas créer d'objets à partir d'elle ?

Une classe abstraite sert de classe de base pour d'autres sous-classes. Ici, QAbstractButton définit un certain nombre de propriétés communes à tous les types de boutons (boutons classiques, cases à cocher, cases radio...). Par exemple, parmi les propriétés communes on trouve :

- **text** : le texte affiché
- **icon** : l'icône affichée à côté du texte du bouton
- **shortcut** : le raccourci clavier pour activer le bouton
- **down** : indique si le bouton est enfoncé ou non
- etc.

Bref, encore une fois tout ça n'est défini qu'une fois dans QAbstractButton, et on le retrouve ensuite automatiquement dans QPushButton, QCheckBox, etc.



Dans ce cas, pourquoi QObject et QWidget ne sont pas des classes abstraites elles aussi ? Après tout, elles ne représentent rien de particulier et servent juste de classes de base !

Oui, vous avez tout à fait raison, leur rôle est d'être des classes de base.

Mais... pour un certain nombre de raisons pratiques (qu'on ne détaillera pas ici), il est possible de les instancier quand même, donc de créer par exemple un objet de type QWidget.

Si on affiche un QWidget, qu'est-ce qui apparaît ? Une fenêtre !

En fait, un widget qui ne se trouve pas à l'intérieur d'un autre widget est considéré comme une fenêtre. Ce qui explique pourquoi, en l'absence d'autre information, Qt décide de créer une fenêtre.

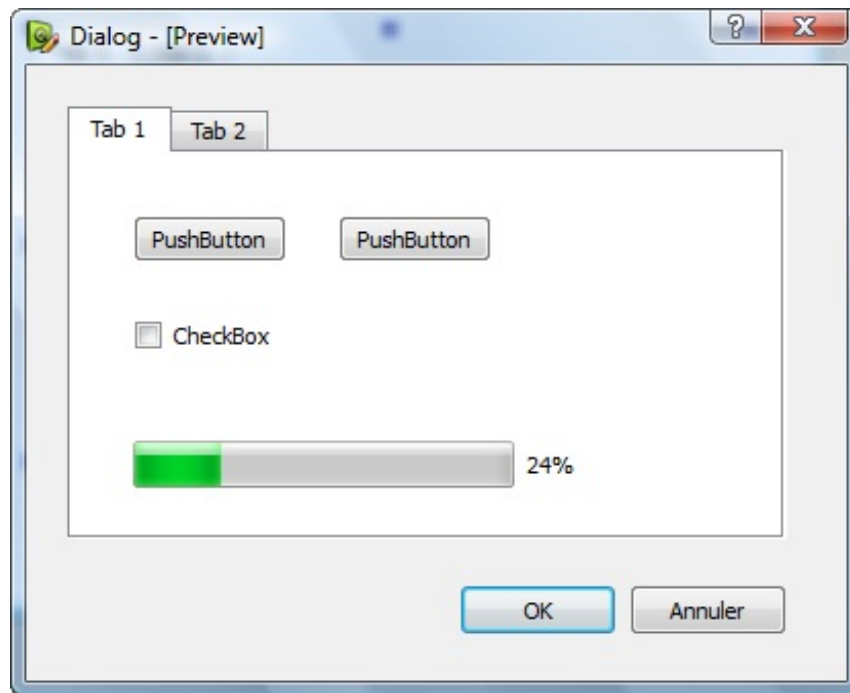
Un widget peut en contenir un autre

Nous attaquons maintenant une notion importante, pas très compliquée, qui est celle des **widgets conteneurs**.

Contenant et contenu

Il faut savoir qu'un widget peut en contenir un autre. Par exemple, une fenêtre (un QWidget) peut contenir 3 boutons (QPushButton), une case à cocher (QCheckBox), une barre de progression (QProgressBar), etc.

Ce n'est pas là de l'héritage, juste une histoire de contenant et de contenu. Prenons un exemple :



Sur cette capture, la fenêtre contient 3 widgets :

- Un bouton OK
- Un bouton Annuler
- Un conteneur avec des onglets

Le conteneur avec des onglets est, comme son nom l'indique, un conteneur. Il contient à son tour des widgets :

- 2 boutons
- Une checkbox
- Une barre de progression

Les widgets sont donc imbriqués les uns dans les autres de cette manière :

- QWidget (la fenêtre)
 - QPushButton
 - QPushButton
 - QWidget (le conteneur à onglets)
 - QPushButton
 - QPushButton
 - QCheckBox
 - QProgressBar



Attention : ne confondez pas ceci avec l'héritage ! Dans cette partie, je suis en train de vous montrer qu'un widget peut en contenir d'autres. Le gros schéma qu'on a vu un peu plus haut n'a rien à voir avec la notion de widget conteneur. Ici, on découvre qu'un widget peut en contenir d'autres, indépendamment du fait que ce soit une classe mère ou une classe fille.

Créer une fenêtre contenant un bouton

On ne va pas commencer par faire une fenêtre aussi compliquée que celle que nous venons de voir. Pour le moment on va s'entraîner à faire quelque chose de simple : créer une fenêtre qui contient un bouton.



Mais... c'est pas ce qu'on a fait tout le temps jusqu'ici ? 🤔

Non, ce qu'on a fait jusqu'ici c'était juste afficher un bouton. Automatiquement, Qt a créé une fenêtre autour car on ne peut pas avoir de bouton qui "flotte" seul sur l'écran.

L'avantage de créer une fenêtre *puis* de mettre un bouton dedans, c'est que :

- On pourra mettre d'autres widgets à l'intérieur de la fenêtre à l'avenir.
- On pourra placer le bouton où on veut dans la fenêtre avec les dimensions qu'on veut (jusqu'ici le bouton avait toujours la même taille que la fenêtre).

Voilà comment il faut faire :

Code : C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

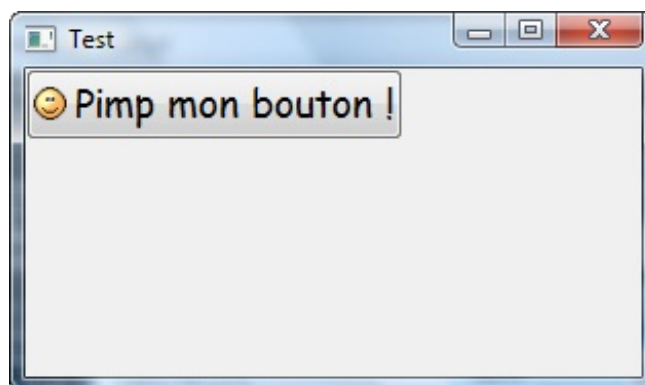
    // Création d'un widget qui servira de fenêtre
    QWidget fenetre;
    fenetre.setFixedSize(300, 150);

    // Création du bouton, ayant pour parent la "fenetre"
    QPushButton bouton("Pimp mon bouton !", &fenetre);
    // Customisation du bouton
    bouton.setFont(QFont("Comic Sans MS", 14));
    bouton.setCursor(Qt::PointingHandCursor);
    bouton.setIcon(QIcon("smile.png"));

    // Affichage de la fenêtre
    fenetre.show();

    return app.exec();
}
```

... et le résultat :



Qu'est-ce qu'on a fait ?

1. On a créé une fenêtre à l'aide d'un objet de type QWidget.

2. On a dimensionné notre widget (donc notre fenêtre) avec la méthode `setFixedSize`. La taille de la fenêtre sera fixée : on ne pourra pas la redimensionner.
3. On a créé un bouton, mais avec cette fois une nouveauté au niveau du constructeur : on a indiqué un pointeur vers le widget parent (en l'occurrence la fenêtre).
4. On a customisé un peu le bouton pour la forme.
5. On a déclenché l'affichage de la fenêtre (et donc du bouton qu'elle contenait).

Tous les widgets possèdent un constructeur surchargé qui permet d'indiquer quel est le parent du widget que l'on crée. Il suffit de donner un pointeur pour que Qt sache "qui contient qui".

Le paramètre "&fenetre" du constructeur permet donc d'indiquer que la fenêtre est le parent de notre bouton :

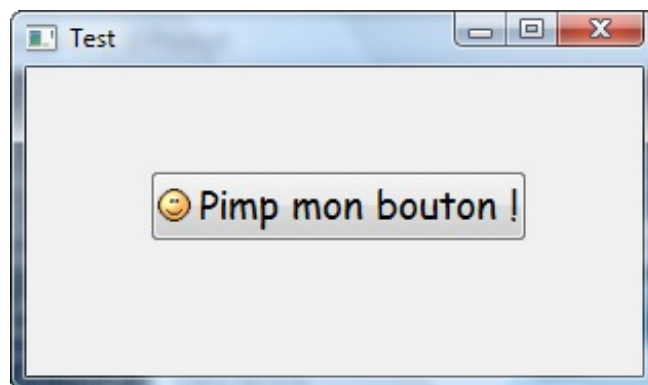
Code : C++

```
QPushButton bouton("Pimp mon bouton !", &fenetre);
```

Si vous voulez placer le bouton ailleurs dans la fenêtre, utilisez la méthode `move` :

Code : C++

```
bouton.move(60, 50);
```



A noter aussi la méthode `setGeometry`, qui prend 4 paramètres :

Code : C++

```
bouton.setGeometry(abscisse, ordonnee, largeur, hauteur);
```

La méthode `setGeometry` permet donc, en plus de déplacer le widget, de lui donner une dimension bien précise.

Tout widget peut en contenir d'autres

... même les boutons !

Quel que soit le widget, son constructeur accepte en dernier paramètre un pointeur vers un autre widget pour indiquer quel est le parent.

On peut faire le test si vous voulez en plaçant un bouton... dans notre bouton !

Code : C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    fenetre.setFixedSize(300, 150);

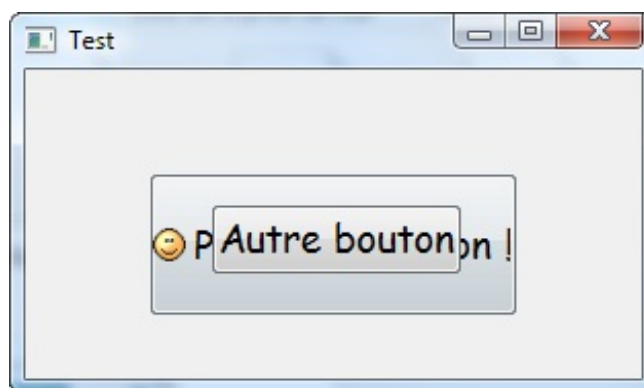
    QPushButton bouton("Pimp mon bouton !", &fenetre);
    bouton.setFont(QFont("Comic Sans MS", 14));
    bouton.setCursor(Qt::PointingHandCursor);
    bouton.setIcon(QIcon("smile.png"));
    bouton.setGeometry(60, 50, 180, 70);

    // Création d'un autre bouton ayant pour parent le premier
    bouton
    QPushButton autreBouton("Autre bouton", &bouton);
    autreBouton.move(30, 15);

    fenetre.show();

    return app.exec();
}
```

Résultat : notre bouton est placé à l'intérieur de l'autre bouton !



Cet exemple montre qu'il est donc possible de placer un widget dans n'importe quel autre widget, même un bouton. Bien entendu, comme le montre ma capture d'écran, ce n'est pas très malin de faire ça, mais ça prouve que Qt est très flexible 😊

Des includes "oubliés"

Dans le code source précédent, nous avons utilisé les classes QWidget, QFont et QIcon pour créer des objets. Normalement, nous devrions faire un include des fichiers headers de ces classes en plus de QPushButton et QApplication pour que le compilateur les connaisse :

Code : C++

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QFont>
#include <QIcon>
```



Ah ben oui ! Si on n'a pas inclus le header de la classe QWidget, comment est-ce qu'on a pu créer tout à l'heure un objet "fenetre" de type QWidget sans que le compilateur ne hurle à la mort ?

Coup de bol. En fait, on avait inclus QPushButton. Et comme QPushButton hérite de QWidget, il avait lui-même inclus QWidget dans son header.

Quant à QFont et QIcon, ils étaient inclus eux aussi car indirectement utilisés par QPushButton.

Bref, des fois comme ça ça marche et on a de la chance. Normalement, si on faisait *très* bien les choses, on devrait faire un include par classe utilisée.

C'est un peu lourd et il m'arrive d'en oublier. Comme ça marche, en général je ne me pose pas trop de questions.

Toutefois, si vous voulez être sûr d'inclure une bonne fois pour toutes toutes les classes du module "Qt GUI", il vous suffit de faire :

Code : C++

```
#include <QtGui>
```

Le header "QtGui" inclut à son tour toutes les classes du module GUI, donc QWidget, QPushButton, QFont, etc. Attention toutefois, la compilation sera un peu ralentie du coup.

Hériter un widget

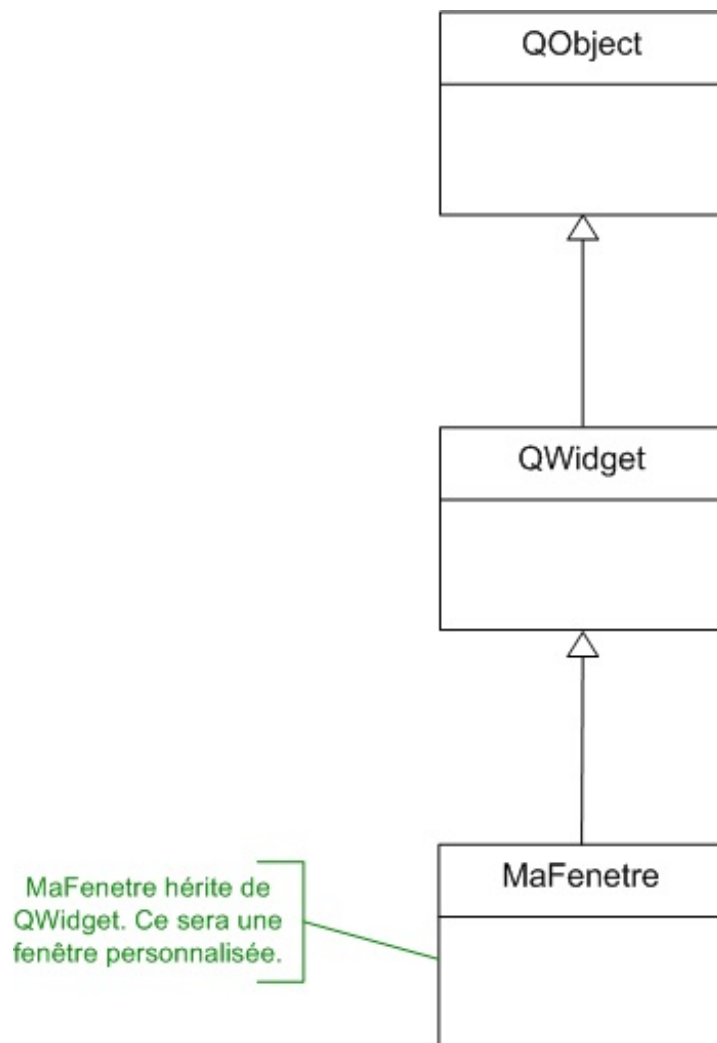
Bon résumons !

Jusqu'ici dans ce chapitre, nous avons :

- Appris à lire et modifier les propriétés d'un widget, en voyant quelques exemples de propriétés des boutons.
- Découvert de quelle façon étaient architecturées les classes de Qt, avec les multiples héritages.
- Découvert la notion de widget conteneur (un widget peut en contenir d'autres). Pour nous entraîner, nous avons créé une fenêtre puis inséré un bouton à l'intérieur.

Nous allons ici aller plus loin dans la personnalisation des widgets en "inventant" un nouveau type de widget. En fait, nous allons créer une nouvelle classe qui va hériter de QWidget et représenter notre fenêtre. Créer une classe pour gérer la fenêtre va peut-être vous paraître un peu lourd au premier abord, mais c'est pourtant comme ça qu'on fait à chaque fois que l'on crée des GUI en POO. Ca nous donnera une plus grande souplesse par la suite.

L'héritage que l'on va faire sera donc le suivant :



Allons-y 😊

Qui dit nouvelle classe dit 2 nouveaux fichiers :

- MaFenetre.h : contiendra la définition de la classe
- MaFenetre.cpp : contiendra l'implémentation des méthodes

Edition des fichiers

MaFenetre.h

Voici le code du fichier MaFenetre.h, nous allons le commenter tout de suite après :

Code : C++

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>

class MaFenetre : public QWidget // On hérite de QWidget (IMPORTANT)
{
public:
    MaFenetre();
```

```
private:
    QPushButton *m_bouton;
};

#endif
```

Quelques petites explications :

Code : C++

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

// Contenu

#endif
```

Là, nous protégeons le header contre les inclusions infinies grâce à cette bonne vieille méthode du `#ifndef`.

Code : C++

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
```

Comme nous allons hériter de `QWidget`, il est nécessaire d'inclure la définition de cette classe.

Par ailleurs, nous allons utiliser un `QPushButton`, donc on inclut le header là aussi.

Quant à `QApplication`, on ne l'utilise pas ici, mais on en aura besoin dans le chapitre suivant, je prépare un peu le terrain 😊

Code : C++

```
class MaFenetre : public QWidget // On hérite de QWidget (IMPORTANT)
{
```

C'est le début de la définition de la classe. Si vous vous souvenez de l'héritage, ce que j'ai fait là ne devrait pas trop vous choquer. Le `: public QWidget` signifie que notre classe hérite de `QWidget`. Nous récupérons donc automatiquement toutes les propriétés de `QWidget`.

Code : C++

```
public:
    MaFenetre();

private:
    QPushButton *m_bouton;
```

Le contenu de la classe est très simple.

Nous écrivons le prototype du constructeur. C'est un prototype minimal (`MaFenetre()`), mais cela nous suffira. Le constructeur est public, car s'il était privé on ne pourrait jamais créer d'objet à partir de cette classe 😊

Nous créons un attribut "m_bouton" de type QPushButton. Notez que celui-ci est un pointeur, il faudra donc le "construire" de manière dynamique avec l'aide du mot-clé new. Tous les attributs devant être privés, nous avons fait précéder cette ligne d'un "private:" qui interdira aux utilisateurs de la classe de modifier directement le bouton.

MaFenetre.cpp

Le fichier .cpp contient l'implémentation des méthodes de la classe. Comme notre classe ne contient qu'une méthode (le constructeur), le fichier .cpp ne sera donc pas long à écrire :

Code : C++

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

    // Construction du bouton
    m_bouton = new QPushButton("Pimp mon bouton !", this);

    m_bouton->setFont(QFont("Comic Sans MS", 14));
    m_bouton->setCursor(Qt::PointingHandCursor);
    m_bouton->setIcon(QIcon("smile.png"));
    m_bouton->move(60, 50);
}
```

Quelques explications :

Code : C++

```
#include "MaFenetre.h"
```

C'est obligatoire pour inclure les définitions de la classe.

Tout ça ne devrait pas être nouveau pour vous, nous avons fait ça de nombreuses fois dans la partie précédente du cours 😊

Code : C++

```
MaFenetre::MaFenetre() : QWidget()
{
```

L'en-tête du constructeur. Il ne faut pas oublier de le faire précéder d'un "MaFenetre::" pour que le compilateur sache à quelle classe celui-ci se rapporte.

Le ": QWidget()" sert à appeler le constructeur de QWidget en premier lieu. Parfois, on en profitera pour envoyer au constructeur de QWidget quelques paramètres, mais là on va se contenter du constructeur par défaut.

Code : C++

```
    setFixedSize(300, 150);
```

Rien d'extraordinaire : on définit la taille de la fenêtre de manière fixée, pour interdire son redimensionnement.

Vous noterez qu'on n'a pas eu besoin d'écrire fenetre.setFixedSize(300, 150);. Pourquoi ? Parce qu'on est dans la classe. On ne fait qu'appeler une des méthodes de la classe (setFixedSize), méthode qui appartient à QWidget, et donc qui appartient aussi à notre

classe puisqu'on hérite de QWidget 😊

J'avoue j'avoue, ce n'est pas évident de bien se repérer au début. Pourtant, vous pouvez me croire, tout ceci est logique mais ça vous paraîtra plus clair à force de pratiquer. Pas de panique donc si vous vous dites "oh mon dieu j'aurais jamais pu deviner ça 😊". Faites-moi confiance c'est tout 😊

Code : C++

```
m_bouton = new QPushButton("Pimp mon bouton !", this);
```

C'est la ligne la plus délicate de ce constructeur.

Ici nous construisons le bouton. En effet, dans le header nous n'avons fait que créer le pointeur, mais il ne pointait vers rien jusqu'ici !

Le new permet d'appeler le constructeur de la classe QPushButton et d'affecter une adresse au pointeur.

Autre détail un tout petit peu délicat : le mot-clé *this*. Je vous en avais parlé dans la partie précédente du cours, en vous disant "faites-moi confiance, même si ça vous paraît inutile maintenant, ça vous sera indispensable plus tard".

Bonne nouvelle : c'est maintenant que vous découvrirez un cas où le mot-clé *this* nous est indispensable ! En effet, le second paramètre du constructeur doit être un pointeur vers le widget parent. Quand nous faisons tout dans le main, c'était simple : il suffisait de donner le pointeur vers l'objet fenêtre. Mais là, **nous sommes dans la fenêtre** ! En effet, nous écrivons la classe MaFenetre. C'est donc "moi", la fenêtre, qui sers de widget parent. Pour donner le pointeur vers moi, il suffit d'écrire le mot-clé *this*.

Et toujours... main.cpp

Bien entendu, que serait un programme sans son main ?

Ne l'oublions pas celui-là !

La bonne nouvelle, c'est que comme bien souvent dans les gros programmes, notre main va être tout petit. Ridiculement petit. Microscopique. Microbique même.

Code : C++

```
#include <QApplication>
#include "MaFenetre.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
}
```

On n'a besoin d'inclure que 2 headers car nous n'utilisons que 2 classes : QApplication et MaFenetre.

Le contenu du main est très simple : on crée un objet de type MaFenetre, et on l'affiche par un appel à la méthode "show()". C'est tout 😊

Lors de la création de l'objet fenetre, le constructeur de la classe MaFenetre est appelé. Dans son constructeur, la fenêtre définit toute seule ses dimensions et les widgets qu'elle contient (en l'occurrence, juste un bouton).

La destruction automatique des widgets enfants



Minute papillon ! On a créé dynamiquement un objet de type QPushButton dans le constructeur de la classe MaFenetre... mais on n'a pas détruit cet objet avec un delete !

En effet, tout objet créé dynamiquement avec un new implique forcément un delete quelque part. Vous avez bien retenu la leçon. Normalement, on devrait écrire le destructeur de MaFenetre, qui contiendrait ceci :

Code : C++

```
MaFenetre::~MaFenetre()  
{  
    delete m_bouton;  
}
```

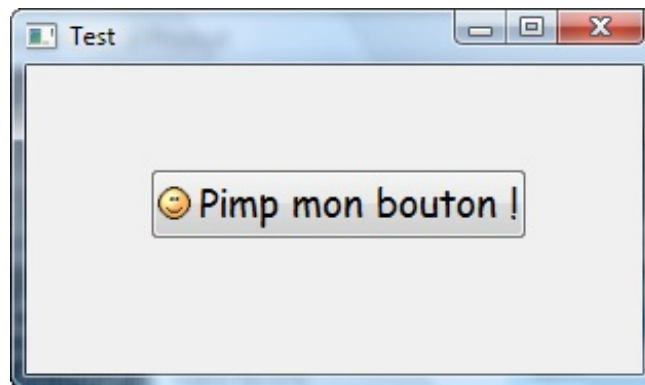
C'est comme ça qu'on doit faire en temps normal. Toutefois, Qt supprimera automatiquement le bouton lors de la destruction de la fenêtre (à la fin du main).

En effet, **quand on supprime un widget parent (ici notre fenêtre), Qt supprime automatiquement tous les widgets qui se trouvent à l'intérieur** (tous les widgets enfants). C'est un des avantages d'avoir dit que le QPushButton avait pour "parent" la fenêtre. Dès qu'on supprime la fenêtre, hop, Qt supprime tout ce qu'elle contient, et donc fait le delete nécessaire du bouton.

Qt nous simplifie la vie en nous évitant d'avoir à écrire tous les delete des widgets enfants. N'oubliez pas néanmoins que tout new implique normalement un delete. Ici, on profite du fait que Qt le fasse pour nous.

Compilation

Le résultat, si tout va bien, devrait être le même que tout à l'heure :



QUOI ? TOUT CE BAZAR POUR FAIRE LA MÊME CHOSE AU FINAL ???



Mais non mais non



En fait, on vient de créer des fondements beaucoup plus solides pour notre fenêtre en faisant ce qu'on vient de faire. On a déjà un peu plus découpé notre code (et avoir un code modulaire, c'est bien !) et on pourra par la suite plus facilement rajouter de nouveaux widgets et surtout... gérer les événements des widgets !

Mais tout ça, vous le découvrirez... dans le prochain chapitre !



Petit exercice : essayez de modifier (ou de surcharger) le constructeur de la classe MaFenetre pour qu'on puisse lui envoyer en paramètre la largeur et la hauteur de la fenêtre à créer. Ainsi, vous pourrez alors définir les dimensions de la fenêtre lors de sa création dans le main.

Nous avançons dans notre découverte de Qt, c'est bien !



Vous commencez à mieux maîtriser le concept de widget et vous avez appris à organiser votre code de manière modulaire afin de servir de base solide pour les chapitres à venir.

Le programme de la suite ? Les signaux et les slots !

Nous allons faire en sorte que notre programme réagisse lorsqu'on clique sur le bouton !

Les signaux et les slots

Nous commençons à maîtriser petit à petit la création d'une fenêtre. Dans le chapitre précédent, nous avons posé de solides bases pour développer par la suite notre application. Nous avons réalisé une classe personnalisée, héritant de QWidget.

Nous allons maintenant découvrir le mécanisme des **signaux et des slots**, un principe propre à Qt qui est clairement un de ses points forts. Il s'agit d'une technique séduisante pour gérer les événements au sein d'une fenêtre.

Par exemple, si on clique sur un bouton, on voudrait qu'une fonction soit appelée pour réagir au clic. C'est précisément ce que nous apprendrons à faire dans ce chapitre, qui va enfin rendre votre application dynamique 😊

Le principe des signaux et slots

Le principe est plutôt simple à comprendre : une application de type GUI réagit à partir d'événements. C'est ce qui rend votre fenêtre dynamique.

Ceux d'entre vous qui ont déjà essayé la bibliothèque SDL se souviennent peut-être de la gestion des événements : interception des touches du clavier, des déplacements de la souris, du joystick, etc.

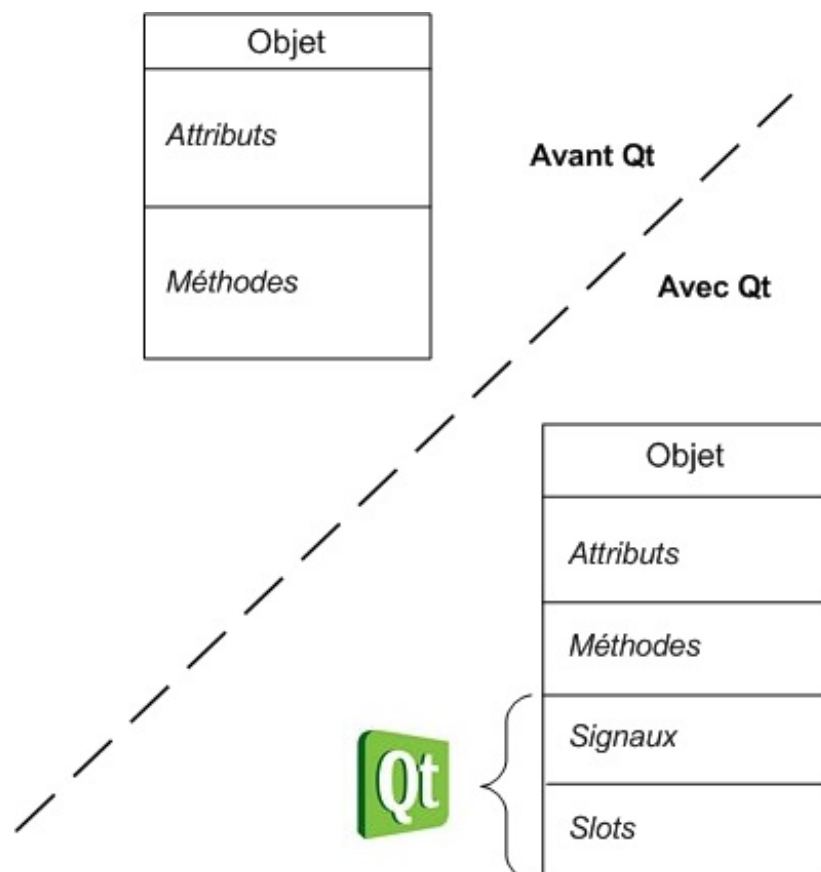
Ce que Qt propose, c'est la même chose mais à plus haut niveau : c'est donc beaucoup plus facile à gérer.

On parle de signaux et de slots, mais qu'est-ce que c'est concrètement ? C'est un concept inventé par Qt. Voici une petite définition en guise d'introduction :

- **Un signal** : c'est un message envoyé par un widget lorsqu'un événement se produit.
Exemple : on a cliqué sur un bouton.
- **Un slot** : c'est la fonction qui est appelée lorsqu'un événement s'est produit. On dit que le signal appelle le slot.
Concrètement, un slot est une méthode d'une classe.
Exemple : le slot quit() de la classe QApplication, qui provoque l'arrêt du programme.

Les signaux et les slots sont considérés par Qt comme des éléments d'une classe à part entière, en plus des attributs et des méthodes.

Voici un schéma qui montre ce qu'un objet pouvait contenir avant Qt, ainsi que ce qu'il peut contenir maintenant qu'on utilise Qt :



Qt rajoute des éléments appelés "Signaux" et "Slots" aux objets

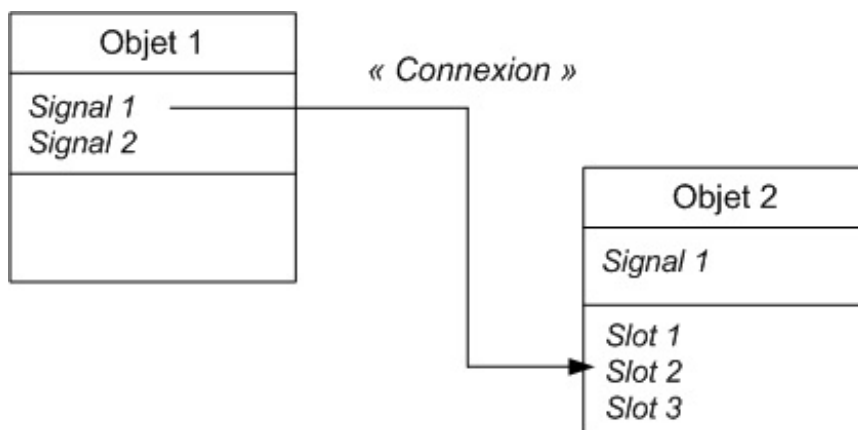
Avant Qt, un objet était constitué d'attributs et de méthodes. C'est tout.

Qt rajoute en plus la possibilité d'utiliser ce qu'il appelle des signaux et des slots pour gérer les événements.

Un signal est un message envoyé par l'objet (par exemple "on a cliqué sur le bouton").

Un slot est une... méthode. En fait, c'est une méthode classique comme toutes les autres, à la différence près qu'elle a le droit d'être connectée à un signal.

Avec Qt, on dit que **l'on connecte des signaux et des slots entre eux**. Supposons que vous ayez deux objets, chacun ayant ses propres attributs, méthodes, signaux et slots (je n'ai pas représenté les attributs et les méthodes sur mon schéma pour simplifier) :



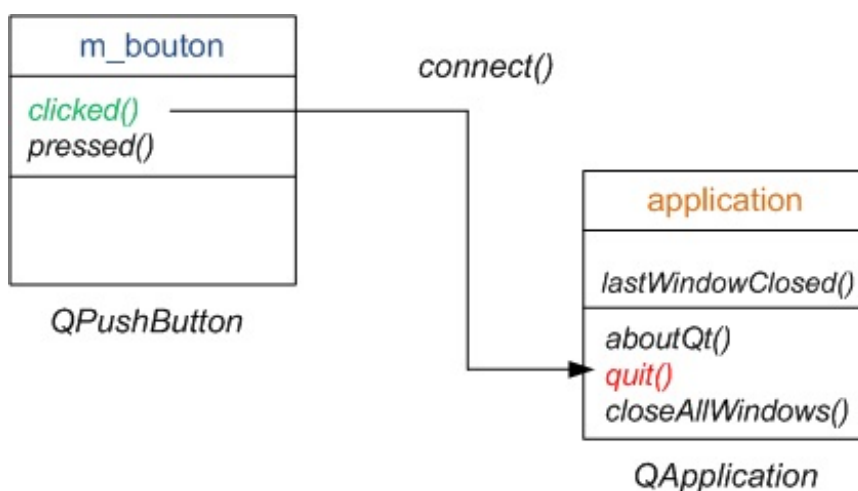
Sur le schéma ci-dessus, on a connecté le signal 1 de l'objet 1 avec le slot 2 de l'objet 2.



Il est possible de connecter un signal à plusieurs slots. Ainsi, un clic sur un bouton pourrait appeler non pas une mais plusieurs méthodes. Comble du raffinement, il est aussi possible de connecter un signal à un autre signal. Le signal d'un bouton peut donc provoquer la création du signal d'un autre widget, qui peut à son tour appeler des slots (voire appeler d'autres signaux pour provoquer une réaction en chaîne !). C'est un peu particulier et on ne verra pas ça dans ce chapitre.

Connexion d'un signal à un slot simple

Voyons un cas très concret. Je vais prendre 2 objets, l'un de type QPushButton, et l'autre de type QApplication. Dans le schéma ci-dessous, ce que vous voyez sont de *vrais signaux et slots* que vous allez pouvoir utiliser :



Regardez attentivement ce schéma. Nous avons d'un côté notre bouton appelé "m_bouton" (de type QPushButton), et de l'autre notre application (de type QApplication, utilisée dans le main).

Nous voudrions par exemple connecter le signal "bouton cliqué" au slot "quitter l'application". Ainsi, un clic sur le bouton provoquerait l'arrêt de l'application.

Pour ce faire, nous devons utiliser une méthode statique de la classe QObject : connect().

Le principe de la méthode connect()

connect() est une méthode statique. Vous vous souvenez ce que ça veut dire ?

Une méthode statique est une méthode d'une classe que l'on peut appeler sans créer d'objet. C'est en fait exactement comme une fonction classique du langage C.



Si vous avez un trou de mémoire, allez vite relire le chapitre traitant des méthodes statiques !

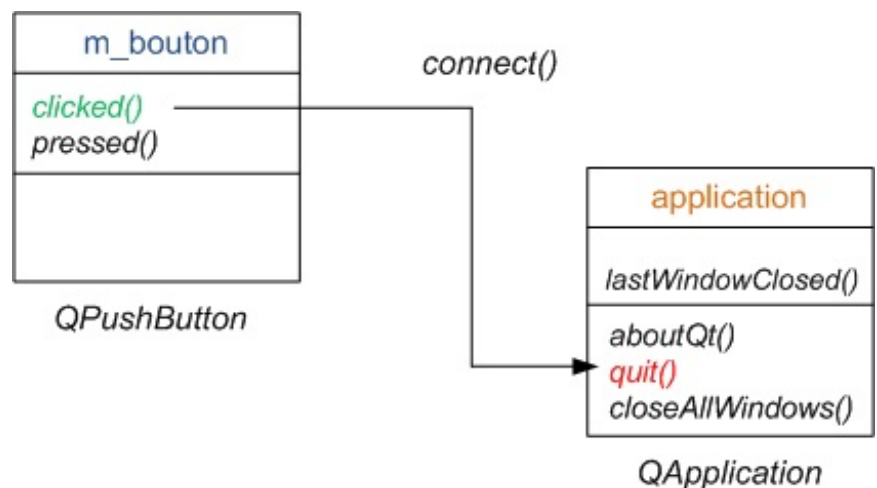
Pour appeler une méthode statique, il faut faire précéder son nom du nom de la classe dans laquelle elle est déclarée. Comme connect() appartient à la classe QObject, il faut donc écrire :

Code : C++

```
QObject::connect ();
```

La méthode connect prend 4 arguments :

- Un pointeur vers l'objet qui émet le signal.
- Le nom du signal que l'on souhaite "intercepter".
- Un pointeur vers l'objet qui contient le slot récepteur.
- Le nom du slot qui doit s'exécuter lorsque le signal se produit.



Pour que vous puissiez vous repérer, j'ai remis ci-contre le schéma qu'on a vu un peu plus haut. Les couleurs sont les mêmes, cela devrait vous permettre de bien visualiser à quoi correspond chaque attribut.

Il existe aussi une méthode disconnect() permettant de casser la connexion entre 2 objets, mais on n'en parlera pas ici car on en a rarement besoin.

Utilisation de la méthode connect() pour quitter

Revenons au code, et plus précisément au constructeur de MaFenetre (fichier MaFenetre.cpp). Ajoutez cette ligne :

Code : C++

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

    m_bouton = new QPushButton("Quitter", this);
    m_bouton->setFont(QFont("Comic Sans MS", 14));
    m_bouton->move(110, 50);

    // Connexion du clic du bouton à la fermeture de l'application
    QObject::connect(m_bouton, SIGNAL(clicked()), qApp, SLOT(quit()));
}
```



`connect()` est une méthode de la classe `QObject`. Comme notre classe `MaFenetre` hérite de `QObject` indirectement, elle possède elle aussi cette méthode. Cela signifie que dans ce cas, et dans ce cas uniquement, on peut enlever le préfixe `QObject::` devant le `connect()` pour appeler la méthode statique.

J'ai choisi de conserver ce préfixe dans le cours pour rappeler qu'il s'agit d'une méthode statique, mais sachez donc qu'il n'a rien d'obligatoire si la méthode est appelée depuis une classe fille de `QObject`.

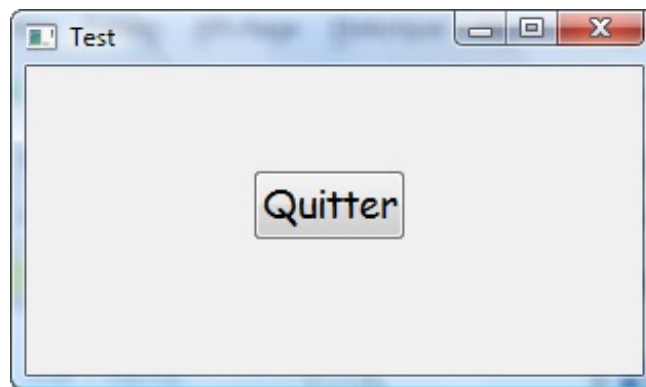
Etudions attentivement cette ligne et plus particulièrement les paramètres que l'on envoie à `connect()` :

- **m_bouton** : c'est un pointeur vers le bouton qui va émettre le signal. Facile.
- **SIGNAL(clicked)** : là c'est assez perturbant comme façon d'envoyer un paramètre. En fait, `SIGNAL()` est une macro du préprocesseur. Qt transformera ça en un code "acceptable" pour la compilation. Le but de cette technique est de vous faire écrire un code court et compréhensible. Ne cherchez pas à comprendre comment Qt fait pour transformer le code, on s'en fout 😊
- **qApp** : c'est un pointeur vers l'objet de type `QApplication` que nous avons créé dans le main. D'où sort ce pointeur ? Euh... joker 😊
En fait, Qt crée automatiquement un pointeur appelé `qApp` vers l'objet de type `QApplication` que nous avons créé. Ce pointeur est défini dans le header `<QApplication>`, que nous avons inclus dans "MaFenetre.h".
- **SLOT(quit)** : c'est le slot qui doit être appelé lorsqu'on a cliqué sur le bouton. Là encore, il faut utiliser la macro `SLOT()` pour que Qt traduise ce code "bizarre" en quelque chose de compilable.

Le slot `quit()` de notre objet de type `QApplication` est un **slot prédéfini**. Il en existe d'autres, comme `aboutQt()` qui affiche une fenêtre "A propos de Qt".

Parfois, pour ne pas dire souvent, les slots prédéfinis par Qt ne nous suffiront pas. Nous apprendrons dans la suite de ce chapitre à créer les nôtres.

Testons notre code ! La fenêtre qui s'ouvre est la suivante :



Rien de bien extraordinaire à première vue. Sauf que... si vous cliquez sur le bouton "Quitter", le programme s'arrête ! Hourra, on vient de réussir à connecter notre premier signal à un slot ! 😊

Utilisation de la méthode `connect()` pour afficher "A propos"

On peut faire un autre essai pour se faire un peu plus la main si vous voulez. Je vous ai parlé d'un autre slot de `QApplication` : `aboutQt()`.

Je vous propose de créer un second bouton qui se chargera d'afficher la fenêtre "A propos de Qt".

Je vous laisse rédiger le code tous seuls comme des grands.

...

...

C'est bon ?

Voici le code final 😊

Code : C++

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

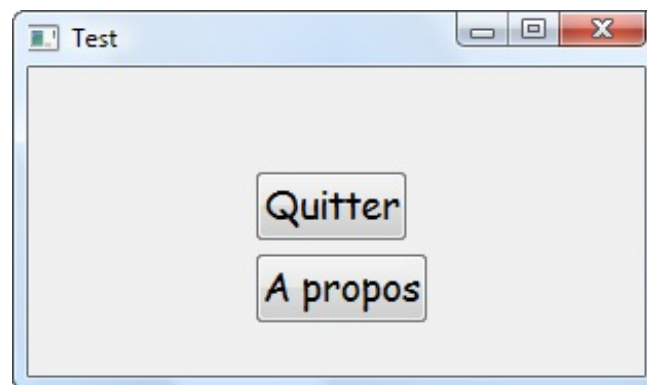
    m_quitter = new QPushButton("Quitter", this);
    m_quitter->setFont(QFont("Comic Sans MS", 14));
    m_quitter->move(110, 50);
    QObject::connect(m_quitter, SIGNAL(clicked()), qApp,
    SLOT(quit()));

    m_aPropos = new QPushButton("A propos", this);
    m_aPropos->setFont(QFont("Comic Sans MS", 14));
    m_aPropos->move(110, 90);
    QObject::connect(m_aPropos, SIGNAL(clicked()), qApp,
    SLOT(aboutQt()));
}
```

Vous noterez que j'ai pris la liberté de nommer les boutons avec des noms un peu plus compréhensibles.

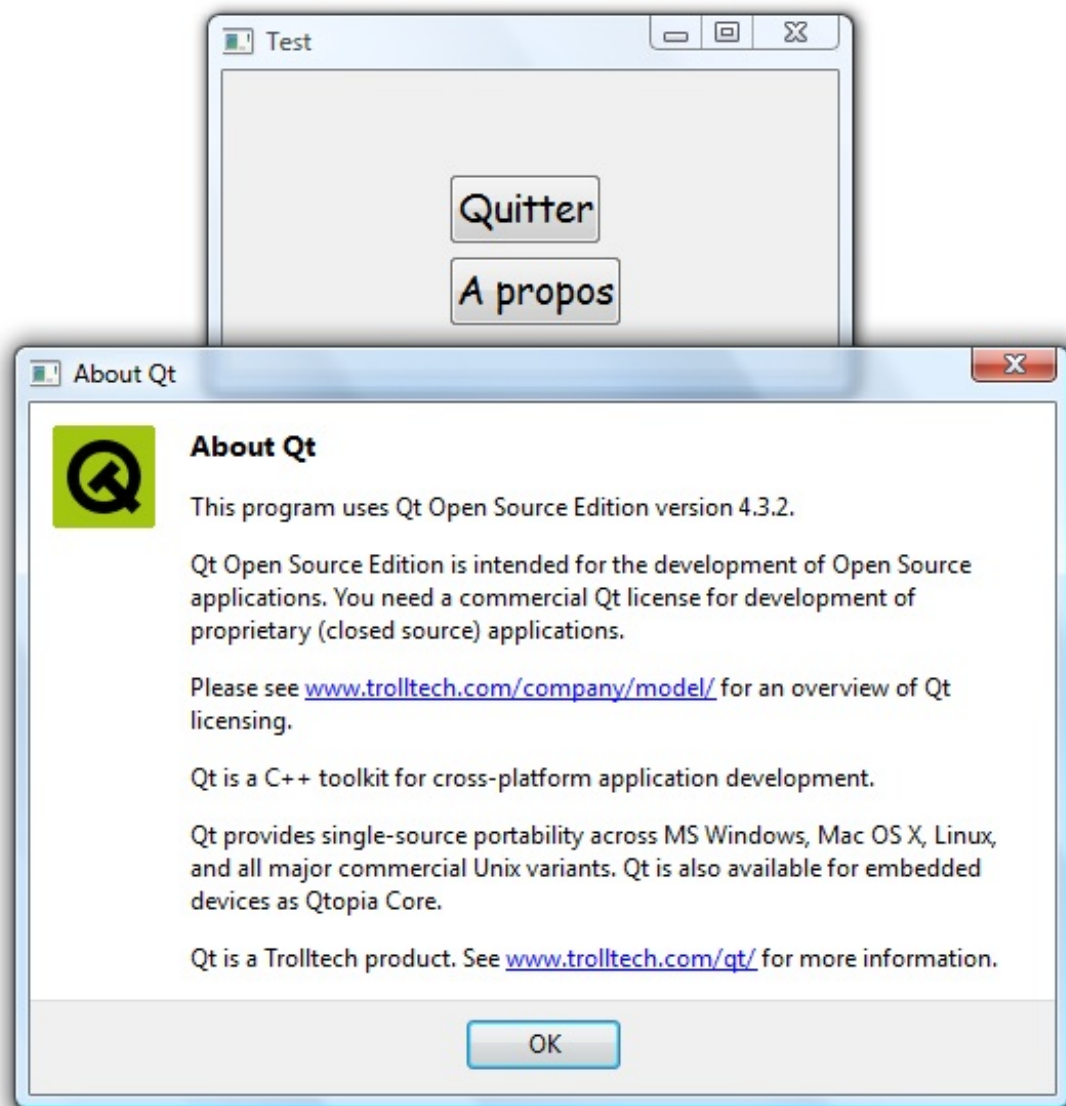
Bien entendu, le fichier MaFenetre.h a un peu changé lui aussi du coup pour déclarer les attributs "m_quitter" et "m_aPropos", mais vous êtes assez grands pour le faire sans moi 😊

Le résultat est une fenêtre qui affiche 2 boutons :



Le bouton "Quitter" ferme toujours l'application.

Quant à "A propos", il provoque l'ouverture de la fenêtre "A propos de Qt".



Des paramètres dans les signaux et slots

La méthode statique `connect()` est assez originale, vous l'avez vu. Il s'agit justement d'une des particularités de Qt que l'on ne retrouve pas dans les autres bibliothèques.

Ces autres bibliothèques, comme `wxWidgets` par exemple, utilisent à la place de nombreuses macros et se servent du mécanisme un peu complexe et délicat des pointeurs de fonction (pour indiquer l'adresse de la fonction à appeler en mémoire).

Il y a d'autres avantages à utiliser la méthode `connect()` avec Qt. On va ici découvrir que **les signaux et les slots peuvent s'échanger des paramètres !**

Dessin de la fenêtre

Dans un premier temps, nous allons placer de nouveaux widgets dans notre fenêtre. Vous pouvez enlever les boutons, on ne va plus s'en servir ici.

A la place, je souhaite vous faire utiliser 2 nouveaux widgets :

- **QSlider** : un curseur qui permet de définir une valeur.
- **QLCDNumber** : un widget qui affiche un nombre.

On va aller un peu plus vite, je vous donne le code directement pour créer ça. Tout d'abord, le header :

Code : C++

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QLCDNumber>
#include <QSlider>

class MaFenetre : public QWidget
{
public:
    MaFenetre();

private:
    QLCDNumber *m_lcd;
    QSlider *m_slider;
};

#endif
```

J'ai donc enlevé les boutons comme vous pouvez le voir, et rajouté un QLCDNumber et un QSlider. Surtout, n'oubliez pas d'inclure le header de ces classes pour pouvoir les utiliser. J'ai gardé l'include du QPushButton ici, ça ne fait pas de mal de le laisser mais si vous ne comptez pas le réutiliser vous pouvez le virer sans crainte.

Et le fichier .cpp :

Code : C++

```
#include "MaFenetre.h"

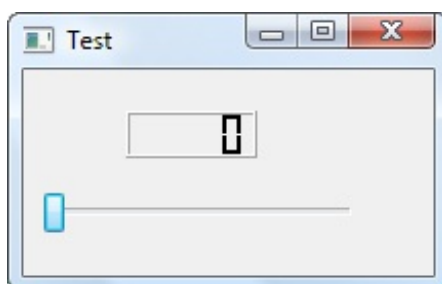
MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(200, 100);

    m_lcd = new QLCDNumber(this);
    m_lcd->setSegmentStyle(QLCDNumber::Flat);
    m_lcd->move(50, 20);

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setGeometry(10, 60, 150, 20);
}
```

Les détails ne sont pas très importants. J'ai modifié le type d'afficheur LCD pour qu'il soit plus lisible (avec setSegmentStyle). Quant au slider, j'ai rajouté un paramètre pour qu'il apparaisse horizontalement (sinon il est vertical).

Voilà qui est fait. Avec ce code, cette petite fenêtre devrait s'afficher :



Connexion avec des paramètres

Maintenant... connexionooooon !

C'est là que les choses deviennent intéressantes. On veut que l'afficheur LCD change de valeur en fonction de la position du curseur du slider.

On dispose du signal et du slot suivant :

- **Le signal `valueChanged(int)` du `QSlider`** : il est émis dès que l'on change la valeur du curseur du slider en le déplaçant. La particularité de ce signal est qu'il envoie un paramètre de type `int` (la nouvelle valeur du slider).
- **Le slot `display(int)` du `QLCDNumber`** : il affiche la valeur qui lui est passée en paramètre.

La connexion se fait avec le code suivant :

Code : C++

```
QObject::connect(m_slider, SIGNAL(valueChanged(int)), m_lcd,  
SLOT(display(int)));
```

Bizarre n'est-ce pas ? 🤔

Il suffit d'indiquer le type du paramètre envoyé, ici un `int`, sans donner de nom à ce paramètre. Qt fait automatiquement la connexion entre le signal et le slot et "transmet" le paramètre au slot.

Le transfert de paramètre se fait comme ceci :

```
QObject::connect(m_slider, SIGNAL(valueChanged(int)), m_lcd, SLOT(display(int)));
```

Ici il n'y a qu'un paramètre à transmettre, c'est donc simple. Sachez toutefois qu'il pourrait très bien y avoir plusieurs paramètres.

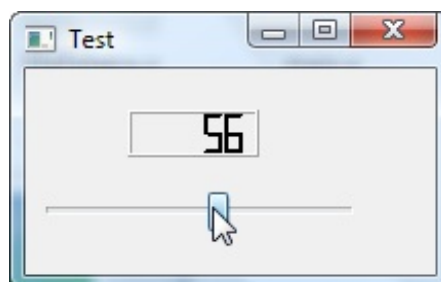


Le type des paramètres doivent correspondre absolument !

Vous ne pouvez pas connecter un signal qui envoie (`int`, `double`) à un slot qui reçoit (`int`, `int`). C'est un des avantages du mécanisme des signaux et des slots : il respecte le type des paramètres. Veillez donc à ce que les signatures soient identiques entre votre signal et votre slot.

En revanche, un signal peut envoyer plus de paramètres à un slot que celui-ci ne peut en recevoir. Dans ce cas, les paramètres supplémentaires seront ignorés.

Résultat : quand on change la valeur du slider, le LCD affiche la valeur correspondante !



Mais comment je sais moi quels sont les signaux et les slots que proposent chacune des classes ? Et aussi, comment je sais qu'un signal envoie un `int` en paramètre ?

La réponse devrait vous paraître simple les amis : **la doc, la doc, la doc !** 🤔

Si vous regardez la [documentation de la classe QLCDNumber](#), vous pouvez voir au début la liste de ses propriétés (attributs) et ses méthodes. Un peu plus bas, vous avez la liste des slots ("Public Slots") et des signaux ("Signals") qu'elle possède !

Les signaux et les slots sont **hérités** comme les attributs et méthodes. Et ça, c'est génial, bien qu'un peu déroutant au début.

Vous noterez donc qu'en plus des [slots propres à QLCDNumber](#), celui-ci propose de nombreux autres slots qui ont été définis dans sa classe parente QWidget, et même des slots issus de QObject ! Vous pouvez par exemple lire :



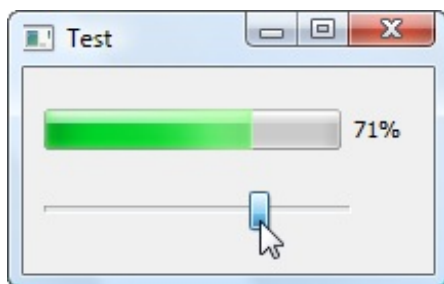
- 19 public slots inherited from QWidget
- 1 public slot inherited from QObject

N'hésitez pas à consulter les slots (ou signaux) qui sont hérités des classes parentes. Parfois on va vous demander d'utiliser un signal ou un slot que vous ne verrez pas dans la page de documentation de la classe : vérifiez donc si celui-ci n'est pas défini dans une classe parente !

Exercice

Pour vous entraîner, je vous propose de réaliser une petite variation du code source précédent.

Au lieu d'afficher le nombre avec un QLCDNumber, affichez-le sous la forme d'une jolie barre de progression comme ceci :



Je ne vous donne que 3 indications qui devraient vous suffire :

- La barre de progression est gérée par un QProgressBar
- Il faut donner des dimensions à la barre de progression pour qu'elle apparaisse correctement, à l'aide de la méthode `setGeometry()` que l'on a déjà vue auparavant.
- Le slot récepteur du QProgressBar est `setValue(int)`. Il s'agit d'un [de ses slots](#), mais la documentation vous indique qu'il y en a d'autres. Par exemple, `reset()` remet à zéro la barre de progression. Pourquoi ne pas ajouter un bouton qui remettrait à zéro la barre de progression ?

C'est tout. Bon courage 😊

Créer ses propres signaux et slots

Voici maintenant une partie très intéressante, bien que plus délicate. Nous allons créer nos propres signaux et slots.

En effet, si en général les signaux et slots par défaut suffisent, il n'est pas rare que l'on se dise "*Zut, le signal (ou le slot) dont j'ai besoin n'existe pas*". C'est dans un cas comme celui-là qu'il devient indispensable de créer son widget personnalisé.



Pour pouvoir créer son propre signal ou slot dans une classe, il faut que celle-ci dérive directement ou indirectement de QObject. C'est le cas de notre classe MaFenetre : elle hérite de QWidget, qui hérite de QObject. On a donc le droit de créer des signaux et des slots dans MaFenetre.

Nous allons commencer par créer notre propre slot, puis nous verrons comment créer notre propre signal.

Créer son propre slot

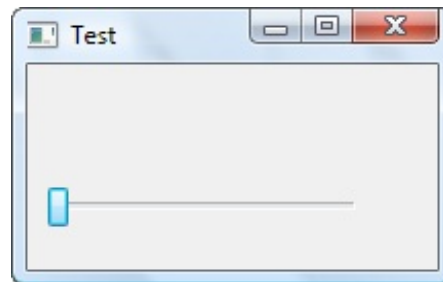
Je vous rappelle tout d'abord qu'un slot n'est rien d'autre qu'une méthode que l'on peut connecter à un signal. Nous allons donc créer une méthode, mais en suivant quelques règles un peu particulières...

Le but du jeu

Pour nous entraîner, nous allons inventer un cas où le slot dont on a besoin n'existe pas.

Je vous propose de conserver le QSlider (je l'aime bien celui-là 😊) et de ne garder que ça sur la fenêtre. Nous allons **faire en sorte que le QSlider contrôle la largeur de la fenêtre**.

Votre fenêtre doit ressembler à cela :



Nous voulons que le signal `valueChanged(int)` du QSlider puisse être connecté à un slot de notre fenêtre (de type `MaFenetre`). Ce nouveau slot aura pour rôle de modifier la largeur de la fenêtre.

Comme il n'existe pas de slot "`changerLargeur`" dans la classe `QWidget`, nous allons devoir le créer.

Pour créer ce slot, il va falloir modifier un peu notre classe `MaFenetre`. Commençons par le header.

Le header (MaFenetre.h)

Dès que l'on doit créer un signal ou un slot personnalisé, il est nécessaire de définir une macro dans le header de la classe.

Cette macro porte le nom de `Q_OBJECT` (tout en majuscules) et doit être placée tout au début de la déclaration de la classe :

Code : C++

```
class MaFenetre : public QWidget
{
    Q_OBJECT

    public:
        MaFenetre();

    private:
        QSlider *m_slider;
};
```

Pour le moment, notre classe ne définit qu'un attribut (le QSlider, privé) et une méthode (le constructeur, public).

La macro `Q_OBJECT` "prépare" en quelque sorte le compilateur à accepter un nouveau mot-clé : "`slot`". Nous allons maintenant pouvoir créer une section "`slots`", comme ceci :

Code : C++

```
class MaFenetre : public QWidget
{
    Q_OBJECT

    public:
        MaFenetre();
```

```
public slots:
void changerLargeur(int largeur);

private:
QSlider *m_slider;
};
```

Vous noterez la nouvelle section "public slots". Je rends toujours mes slots publics. On peut aussi les mettre privés mais ils seront quand même accessibles de l'extérieur car Qt a besoin de pouvoir appeler un slot depuis n'importe quel autre widget.

A part ça, le prototype de notre slot-méthode est tout à fait classique. Il ne nous reste plus qu'à l'implémenter dans le .cpp.

L'implémentation (MaFenetre.cpp)

L'implémentation est d'une simplicité redoutable. Regardez :

Code : C++

```
void MaFenetre::changerLargeur(int largeur)
{
    setFixedSize(largeur, 100);
}
```

Le slot prend en paramètre un entier : la nouvelle largeur de la fenêtre.

Il se contente d'appeler la méthode setFixedSize de la fenêtre et de lui envoyer la nouvelle largeur qu'il a reçue.

Connexion

Bien, voilà qui est fait. Enfin presque : il faut encore connecter notre QSlider au slot de notre fenêtre. Où va-t-on faire ça ? Dans le constructeur de la fenêtre (toujours dans MaFenetre.cpp) :

Code : C++

```
MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(200, 100);

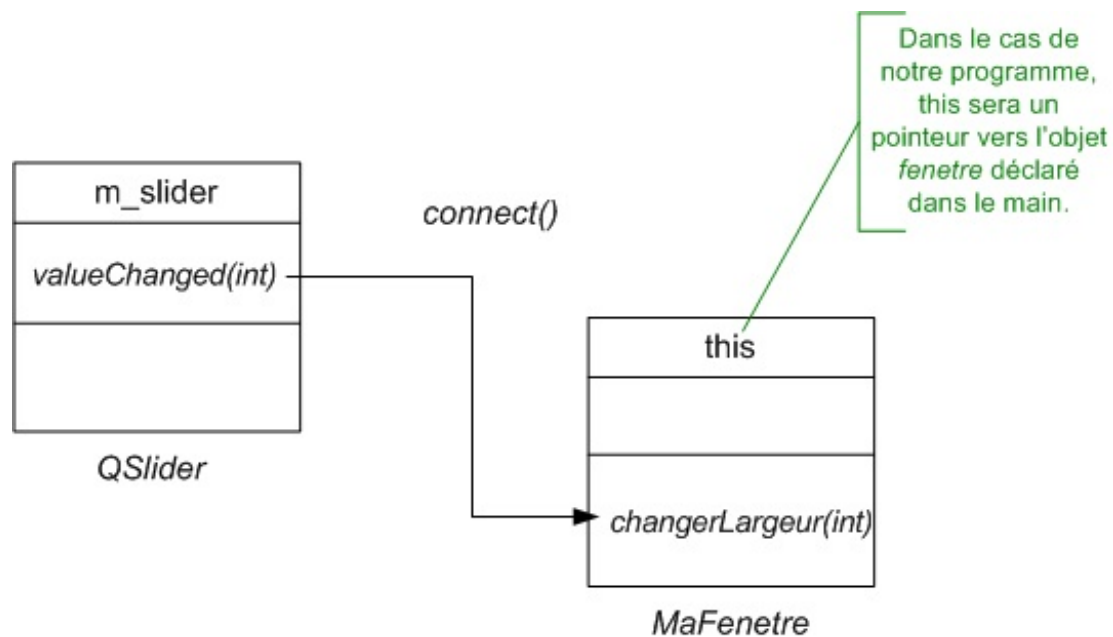
    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setRange(200, 600);
    m_slider->setGeometry(10, 60, 150, 20);

    QObject::connect(m_slider, SIGNAL(valueChanged(int)), this,
        SLOT(changerLargeur(int)));
}
```

J'ai volontairement modifié les différentes valeurs que peut prendre notre slider pour le limiter entre 200 et 600 avec la méthode setRange(). Ainsi, on est sûr que notre fenêtre ne pourra ni être plus petite que 200 pixels de largeur, ni être plus grande que 600 pixels de largeur.

La connexion se fait entre le signal valueChanged(int) de notre QSlider, et le slot changerLargeur(int) de notre classe MaFenetre. Vous voyez là encore un exemple où *this* est indispensable : il faut pouvoir indiquer un pointeur vers l'objet actuel (la fenêtre) et seul *this* peut faire ça !

Schématiquement, on a réalisé la connexion suivante :



Compilation

Avec toutes les nouveautés que nous venons d'utiliser par rapport au C++, la compilation par un make ne suffira pas.

Je vous avais dit qu'il fallait refaire un qmake à chaque fois que les fichiers du projet changeaient. En fait j'ai un peu menti 😊 Comme vous utilisez la macro `Q_OBJECT`, Qt a besoin d'appeler un pré-compilateur qui lui est propre appelé le **moc** (Meta-Object Compiler).

Rassurez-vous, vous n'avez rien à faire de spécial. Relancez juste un qmake avant de faire votre make, et Qt fera le travail de "traduction" du slot en quelque chose de compréhensible pour le compilateur C++.

Vous noterez que le qmake a provoqué la création d'un fichier intermédiaire `moc_MaFenetre.cpp`, ce qui est parfaitement normal. Ce fichier fournit des informations indispensables au compilateur.

Vous pouvez ensuite faire un make, la compilation devrait bien se passer.

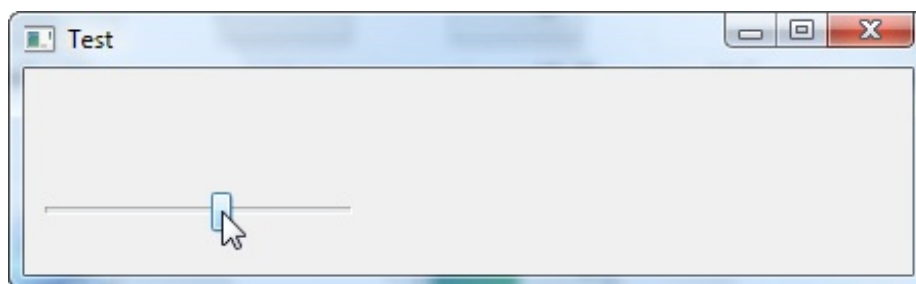


Souvenez-vous ! Si jamais lors de la compilation vous rencontrez l'erreur suivante :

undefined reference to 'vtable for MaFenetre'

... cela signifie que vous n'avez pas fait de qmake avant. Si le moc ne s'est pas exécuté auparavant, la compilation échouera.

Vous pouvez enfin admirer le résultat. Ouf! 😊

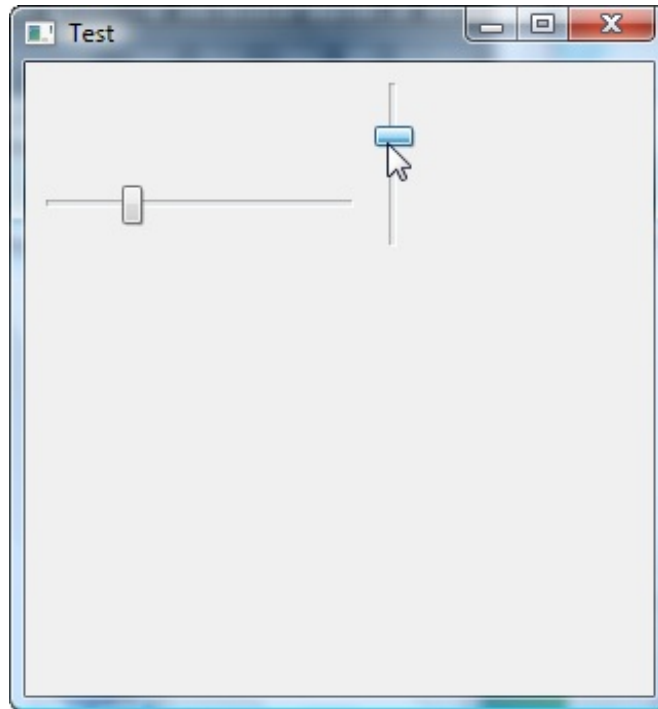


Amusez-vous à redimensionner la fenêtre comme bon vous semblera avec le slider. Comme nous avons fixé les limites du slider entre 200 et 600, la largeur de la fenêtre restera comprise entre 200 et 600 pixels.

Exercice : redimensionner la fenêtre en hauteur

Voici un petit exercice, mais qui va vous forcer à travailler (bande de fainéants, vous me regardez faire depuis tout à l'heure 🤪). Je vous propose de créer un second QSlider, vertical cette fois, qui contrôlera la hauteur de la fenêtre. Pensez à bien définir des limites appropriées pour les valeurs de ce nouveau slider.

Vous devriez obtenir un résultat qui ressemblera à ça :



Si vous voulez "conserver" la largeur pendant que vous modifiez la hauteur, et inversement, vous aurez besoin d'utiliser les méthodes accesseur `width()` (largeur actuelle) et `height()` (hauteur actuelle).

Vous comprendrez très certainement l'intérêt de ces informations lorsque vous coderez. Au boulot !

Créer son propre signal

Il est plus rare d'avoir à créer son signal que son slot, mais cela peut arriver.

Je vous propose de réaliser le programme suivant : si le slider horizontal arrive à sa valeur maximale (600 dans notre cas), alors on émet un signal "agrandissementMax". Notre fenêtre doit pouvoir émettre l'information comme quoi elle est agrandie au maximum. Après, nous connecterons ce signal à un slot pour vérifier que notre programme réagit correctement.

Le header (*MaFenetre.h*)

Commençons par changer le header :

Code : C++

```
class MaFenetre : public QWidget
{
    Q_OBJECT

public:
    MaFenetre();

public slots:
```



```
void changerLargeur(int largeur);

signals:
void agrandissementMax();

private:
QSlider *m_slider;
};
```

On a ajouté une section "signals". Les signaux se présentent en pratique sous forme de méthodes (comme les slots) à la différence près qu'on ne les implémente pas dans le .cpp. En effet, c'est Qt qui le fait pour nous. Si vous tentez d'implémenter un signal, vous aurez une erreur du genre "Multiple definition of...".

Un signal peut passer un ou plusieurs paramètres. Dans notre cas, il n'en envoie aucun.
Un signal doit toujours renvoyer void.

L'implémentation (MaFenetre.cpp)

Maintenant que notre signal est défini, il faut que notre classe puisse l'émettre à un moment.
Quand est-ce qu'on sait que la fenêtre a été agrandie au maximum ? Dans le slot changerLargeur ! Il suffit de tester dans ce slot si la largeur correspond au maximum (600), et d'émettre alors le signal "Youhou, j'ai été agrandie au maximum !".

Retournons dans MaFenetre.cpp et implémentons ce test qui émet le signal depuis changerLargeur :

Code : C++

```
void MaFenetre::changerLargeur(int largeur)
{
    setFixedSize(largeur, height());

    if (largeur == 600)
    {
        emit agrandissementMax();
    }
}
```

Notre méthode s'occupe toujours de redimensionner la fenêtre, mais vérifie en plus si la largeur a atteint le maximum (600). Si c'est le cas, elle émet le signal agrandissementMax().

Pour émettre un signal, on utilise le mot-clé emit, là encore un terme inventé par Qt qui n'existe pas en C++. L'avantage est que c'est très lisible, on comprend "Emettre le signal agrandissementMax()".



Ici, notre signal n'envoie pas de paramètres. Toutefois, sachez que si vous voulez envoyer un paramètre c'est très simple. Il suffit d'appeler votre signal comme ceci : emit monSignal(parametre1, parametre2, ...);

Connexion

Il ne nous reste plus qu'à connecter notre nouveau signal à un slot. Vous pouvez connecter ce signal au slot que vous voulez. Personnellement, je propose de le connecter à l'application (à l'aide du pointeur global QApplication) pour provoquer l'arrêt du programme.

Ca n'a pas trop de sens je suis d'accord, mais c'est juste pour s'entraîner et vérifier que ça fonctionne. Vous aurez l'occasion de faire des connexions plus logiques plus tard, je ne m'en fais pas pour ça 😊

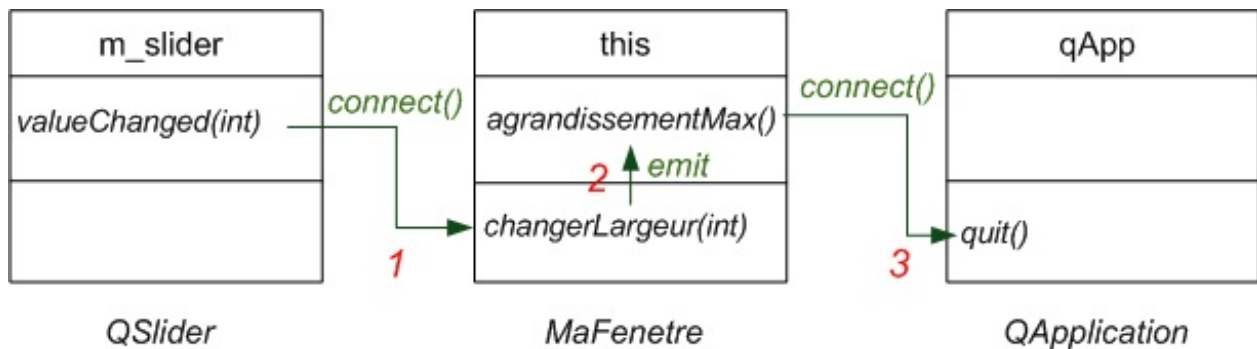
Dans le constructeur de MaFenetre, je rajoute donc :

Code : C++

```
QObject::connect(this, SIGNAL(agrandissementMax()), qApp,
SLOT(quit()));
```

Vous pouvez tester le résultat : normalement le programme s'arrête quand la fenêtre est agrandie au maximum.

Le schéma des signaux qu'on vient d'émettre et connecter est le suivant :



Dans l'ordre, voici ce qui s'est passé :

1. Le signal `valueChanged` du slider a appelé le slot `changerLargeur` de la fenêtre.
2. Le slot a fait ce qu'il avait à faire (changer la largeur de la fenêtre) et a vérifié si la fenêtre était arrivée à sa taille maximale. Lorsque cela a été le cas, le signal personnalisé `agrandissementMax()` a été émis.
3. Le signal `agrandissementMax()` de la fenêtre était connecté au slot `quit()` de l'application, ce qui a provoqué la fermeture du programme.

Et voilà comment le déplacement du slider peut, par réaction en chaîne, provoquer la fermeture du programme ! Bien entendu, ce schéma peut être aménagé et complexifié selon les besoins de votre application.

Maintenant que vous savez créer vos propres slots et signaux, vous avez toute la souplesse nécessaire pour **faire ce que vous voulez** ! 😊

Eh ben dites donc les amis, que de nouveautés dans ce chapitre décidément !

Les signaux et les slots, c'est vraiment ce qui fait la force de Qt... mais ses détracteurs disent que c'est une erreur d'avoir voulu "modifier" le langage C++. En effet, la compilation est plus lourde car il y a des étapes de pré-compilation à effectuer impérativement si on veut que le code soit compilable. C'est un point de vue qui se défend.

L'avantage de ce système, et ça personne ne le discute, c'est qu'il est robuste. On dispose d'une extraordinaire souplesse pour faire communiquer des objets entre eux :

- Un signal peut appeler le slot d'un autre objet pour l'informer d'un événement.
- Un signal peut appeler plusieurs slots d'objets différents si nécessaire pour faire plusieurs traitements.
- Un signal peut être connecté à un autre signal directement, qui lui-même peut être raccordé à un autre signal (réaction en chaîne) ou appeler un slot.
- La connexion entre un signal et un slot permet d'échanger un ou plusieurs paramètres.
- L'échange de paramètres entre le signal et le slot est sécurisé : Qt vérifie que la signature du signal correspond bien à celle du slot.

Les autres bibliothèques, comme `wxWidgets`, utilisent un ensemble de macros, moins lisibles mais qui ne nécessitent pas l'utilisation d'outils intermédiaires comme le moc.

Bref, profitez à fond des signaux et des slots, avec ça vous pouvez vraiment faire ce que vous voulez 😊

Les boîtes de dialogue usuelles

Après un chapitre sur les signaux et les slots riche en nouveaux concepts, on relâche ici un peu la pression. Nous allons découvrir les **boîtes de dialogue usuelles**, aussi appelées "*common dialogs*" par nos amis anglophones.

Qu'est-ce qu'une boîte de dialogue usuelle ? C'est une fenêtre qui sert à remplir une fonction bien précise. Par exemple, on connaît la boîte de dialogue "message" qui affiche un message et ne vous laisse d'autre choix que de cliquer sur le bouton OK. Ou encore la boîte de dialogue "ouvrir un fichier", "enregistrer un fichier", "sélectionner une couleur", etc. On ne s'amuse pas à recréer "à la main" ces fenêtres à chaque fois. On profite de fonctions système pour ouvrir des boîtes de dialogue pré-construites.

Qt s'adapte à l'OS pour afficher une boîte de dialogue qui corresponde aux formes habituelles de votre OS.

En clair : attendez-vous à un chapitre simple qui vous donnera de nombreux outils pour pouvoir interagir avec l'utilisateur de votre programme !

Afficher un message

Le premier type de boîte de dialogue que nous allons voir est le plus courant : la boîte de dialogue "afficher un message".

Nous allons créer un bouton sur notre fenêtre de type `MaFenetre` qui appellera un slot personnalisé. Ce slot ouvrira la boîte de dialogue. En clair, un clic sur le bouton doit pouvoir ouvrir la boîte de dialogue.

Les boîtes de dialogue "afficher un message" sont contrôlées par la classe **`QMessageBox`**. Vous pouvez commencer par faire l'include correspondant dans "`MaFenetre.h`" pour ne pas l'oublier : `#include <QMessageBox>`.

Quelques rappels et préparatifs

Pour que l'on soit sûr de travailler ensemble sur le même code, je vous donne le code source des fichiers `MaFenetre.h` et `MaFenetre.cpp` sur lesquels je vais travailler. Ils ont été simplifiés au maximum histoire d'éviter le superflu.

Code : C++

```
// MaFenetre.h

#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QMessageBox>

class MaFenetre : public QWidget
{
    Q_OBJECT

public:
    MaFenetre();

public slots:
    void ouvrirDialogue();

private:
    QPushButton *m_boutonDialogue;
};

#endif
```

Code : C++

```
// MaFenetre.cpp

#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(230, 120);

    m_boutonDialogue = new QPushButton("Ouvrir la boîte de
dialogue", this);
    m_boutonDialogue->move(40, 50);

    QObject::connect(m_boutonDialogue, SIGNAL(clicked()), this,
SLOT(ouvrirDialogue()));
}

void MaFenetre::ouvrirDialogue()
{
    // Vous insérerez le code d'ouverture des boîtes de dialogue
    ici
}
```

C'est très simple. Nous avons créé un bouton dans la boîte de dialogue qui appelle le slot personnalisé ouvrirDialogue(). C'est dans ce slot que nous nous chargerons d'ouvrir une boîte de dialogue.

Au cas où certains se poseraient la question, notre main.cpp n'a pas changé. Allez, je vous le redonne. Je suis trop sympa je sais, ne me remerciez pas 🤖

Code : C++

```
// main.cpp

#include <QApplication>
#include "MaFenetre.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
}
```

Ouvrir une boîte de dialogue avec une méthode statique

Bien, place à l'action maintenant !

La classe QMessageBox permet de créer des objets de type QMessageBox (comme toute classe qui se respecte 🤖) mais on utilise majoritairement ses méthodes statiques pour des raisons de simplicité. Nous commencerons donc par découvrir les méthodes statiques, qui se comportent je le rappelle comme de simples fonctions. Elles ne nécessiteront pas de créer d'objet.

QMessageBox::information

La méthode statique information() permet d'ouvrir une boîte de dialogue constituée d'une icône "information".

Son prototype est le suivant :

Code : C++

```
StandardButton information ( QWidget * parent, const QString &
title, const QString & text, StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton );
```

Seuls les 3 premiers paramètres sont obligatoires, les autres ayant comme vous le voyez une valeur par défaut. Ces 3 premiers paramètres sont :

- **parent** : un pointeur vers la fenêtre parente (qui doit être de type QWidget ou hériter de QWidget). Vous pouvez envoyer NULL en paramètre si vous ne voulez pas que votre boîte de dialogue ait une fenêtre parente, mais ce sera plutôt rare.
- **title** : le titre de la boîte de dialogue (affiché en haut de la fenêtre).
- **text** : le texte affiché au sein de la boîte de dialogue.

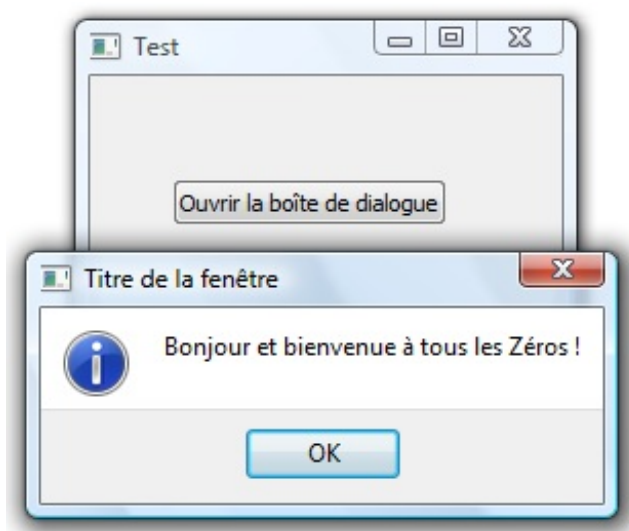
Testons donc un code très simple. Voici le code du slot ouvrirDialogue() :

Code : C++

```
void MaFenetre::ouvrirDialogue ()
{
    QMessageBox::information(this, "Titre de la fenêtre", "Bonjour et
bienvenue à tous les Zéros !");
}
```

L'appel de la méthode statique se fait donc comme celui d'une fonction classique, à la différence près qu'il faut mettre en préfixe le nom de la classe dans laquelle elle est définie (d'où le "QMessageBox:" avant).

Le résultat est une boîte de dialogue comme vous avez l'habitude d'en voir, constituée d'un bouton OK :



Vous noterez que lorsque la boîte de dialogue est ouverte, on ne peut plus accéder à sa fenêtre parente qui est derrière. On dit que la boîte de dialogue est une **fenêtre modale** : c'est une fenêtre qui "bloque" temporairement son parent en attente d'une réponse de l'utilisateur.

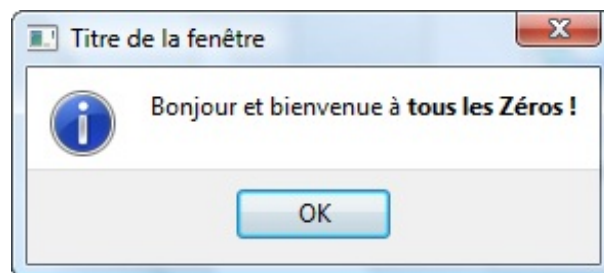
A l'inverse, on dit qu'une fenêtre est **non modale** quand on peut toujours accéder à la fenêtre derrière. C'est le cas en général des boîtes de dialogue "Rechercher un texte" dans les éditeurs de texte.

Comble du raffinement (j'aime bien cette expression 😊), il est même possible de mettre en forme son message à l'aide de balises (X)HTML pour ceux qui connaissent. Si vous ne connaissez pas, il est toujours temps d'[apprendre le HTML](#), j'ai fait un tuto il faut en profiter 😊

Exemple de boîte de dialogue "enrichie" avec du code HTML :

Code : C++

```
QMessageBox::information(this, "Titre de la fenêtre", "Bonjour et  
bienvenue à <strong>tous les Zéros !</strong>");
```



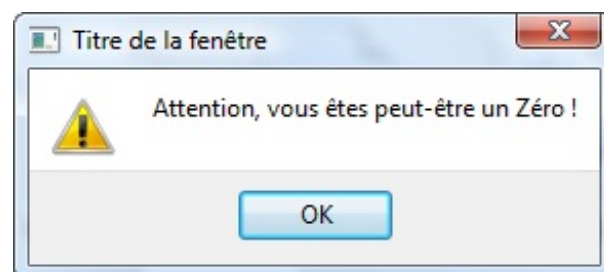
QMessageBox::warning

Si la boîte de dialogue "information" sert à informer l'utilisateur par un message, la boîte de dialogue warning le met en garde contre quelque chose. Elle est généralement accompagné d'un "ding" caractéristique.

Elle s'utilise de la même manière que QMessageBox::information, mais cette fois l'icône change :

Code : C++

```
QMessageBox::warning(this, "Titre de la fenêtre", "Attention, vous  
êtes peut-être un Zéro !");
```



QMessageBox::critical

Quand c'est trop tard et qu'une erreur s'est produite, il ne vous reste plus qu'à utiliser la méthode statique critical() :

Code : C++

```
QMessageBox::critical(this, "Titre de la fenêtre", "Vous n'êtes pas
```

```
un Zéro, sortez d'ici ou j'appelle la police !");
```

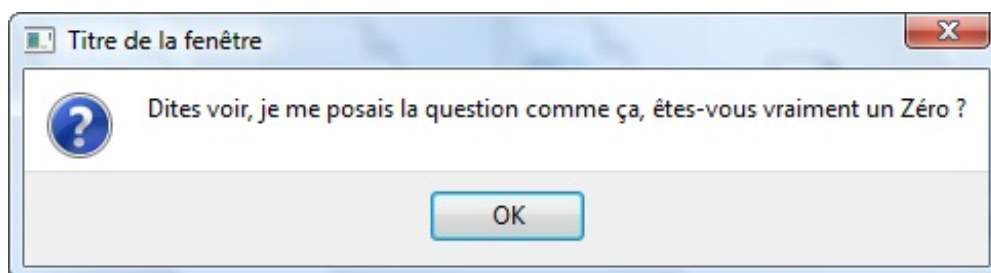


QMessageBox::question

Si vous avez une question à poser à l'utilisateur, c'est la boîte de dialogue qu'il vous faut !

Code : C++

```
QMessageBox::question(this, "Titre de la fenêtre", "Dites voir, je  
me posais la question comme ça, êtes-vous vraiment un Zéro ?");
```



C'est bien joli mais... comment peut-on répondre à la question avec un simple bouton OK ?

Par défaut, c'est toujours un bouton OK qui s'affiche. Mais dans certains cas, comme lorsqu'on pose une question, il faudra afficher d'autres boutons pour que la boîte de dialogue ait du sens.

Personnaliser les boutons de la boîte de dialogue

Pour personnaliser les boutons de la boîte de dialogue, il faut utiliser le 4ème paramètre de la méthode statique. Ce paramètre accepte une combinaison de valeurs prédéfinies, séparées par un OR (la barre verticale |). On appelle cela des *flags*. Si vous avez déjà travaillé avec la SDL, vous connaissez cela. Sinon, vous vous y habituerez vite vous verrez, c'est juste une façon pratique d'envoyer des options à une fonction.



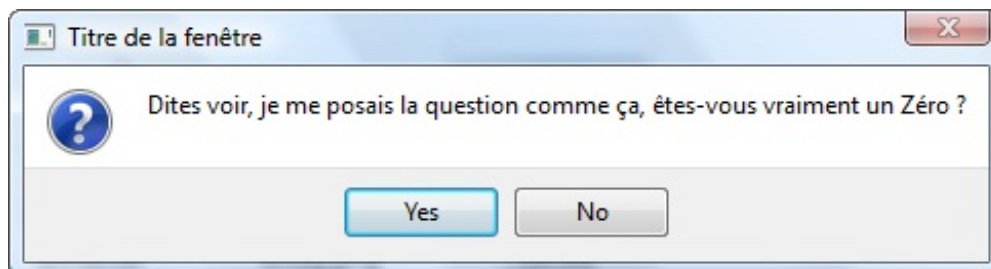
Pour ceux qui se poseraient la question, le 5ème et dernier paramètre de la fonction permet d'indiquer quel est le bouton par défaut. On change rarement cette valeur car Qt choisit généralement le bouton qui convient le mieux par défaut.

La [liste des flags disponibles](#) est donnée par la documentation. Vous avez du choix comme vous pouvez le voir. Si on veut placer les boutons "Oui" et "Non", il nous suffit de combiner les valeurs "QMessageBox::Yes" et "QMessageBox::No"

Code : C++

```
QMessageBox::question(this, "Titre de la fenêtre", "Dites voir, je  
me posais la question comme ça, êtes-vous vraiment un Zéro ?",  
QMessageBox::Yes | QMessageBox::No);
```

Les boutons apparaissent alors :



Horreur ! Malédiction ! Enfer et damnation !

L'anglais me poursuit, les boutons sont écrits en anglais. Catastrophe qu'est-ce que je vais faire au secouuuuuurs !!!

En effet, les boutons sont écrits en anglais. Mais ce n'est pas grave du tout, les applications Qt peuvent être facilement traduites, je vous en avais parlé en introduction de cette partie.

On ne va pas rentrer dans les détails du fonctionnement de la traduction, on aura l'occasion d'en reparler plus longuement plus tard. Je vais vous donner un code à placer dans le fichier `main.cpp`, et vous allez l'utiliser gentiment sans poser de questions. Attention, j'ai dit : sans poser de question. On n'aime pas trop les gens qui posent des questions ici. Un accident est si vite arrivé... 🐼

Code : C++

```
// main.cpp

#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include "MaFenetre.h"

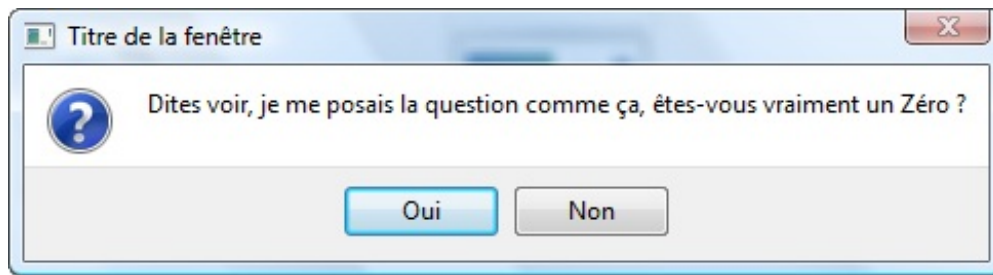
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);
    QTranslator translator;
    translator.load(QString("qt ") + locale,
        QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
}
```

Les lignes ajoutées ont été surlignées. Il y a plusieurs includes et quelques lignes de code supplémentaires dans le `main`. Normalement, votre application devrait maintenant afficher des boutons en français :



Et voilà le travail ! 😊



C'est cool, mais comment je fais pour savoir sur quel bouton l'utilisateur a cliqué ? Hein, hein ?

Quoi ? Encore une question ?

Vous savez, vous réduisez votre espérance de vie avec toutes les questions que vous posez aujourd'hui. Enfin moi j'dis ça comme ça 🤔

Bon ok, cette question est pertinente, je peux y répondre. Je dois y répondre même. Alors allons-y !

Récupérer la valeur de retour de la boîte de dialogue

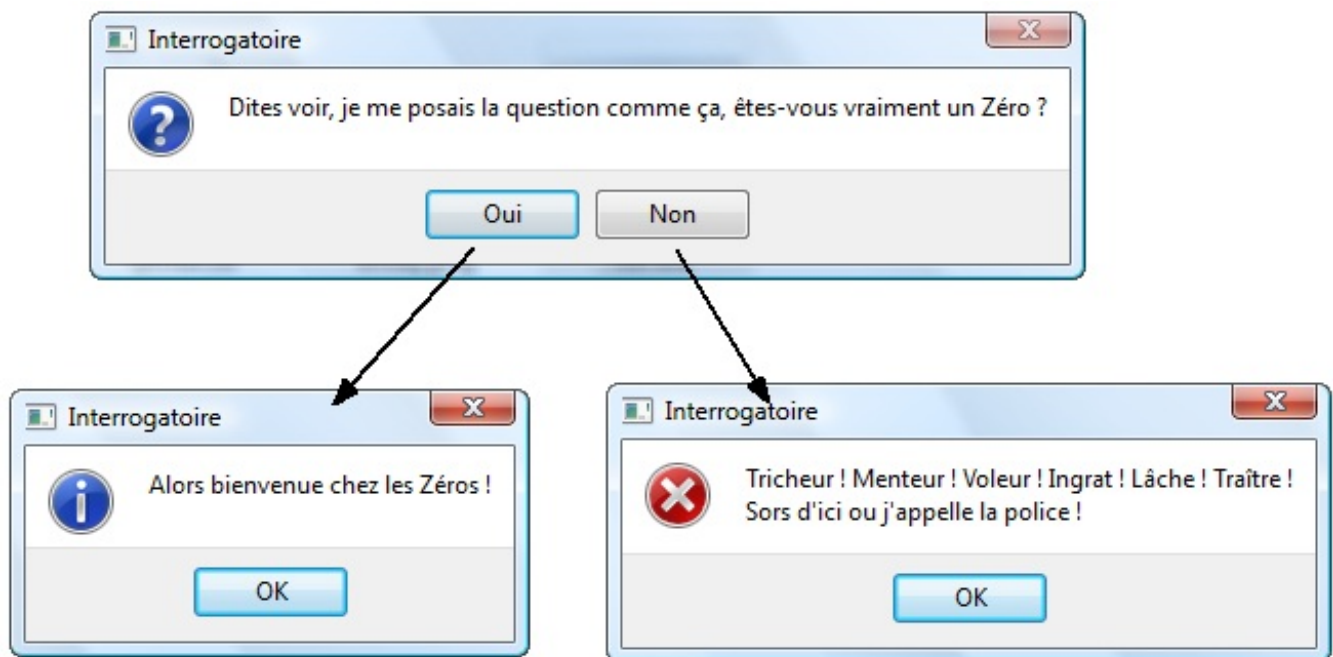
Les méthodes statiques que nous venons de voir retournent un entier (int). On peut tester facilement la signification de ce nombre à l'aide des valeurs prédéfinies par Qt (comme quoi les énumérations c'est pratique !).

Code : C++

```
void MaFenetre::ouvrirDialogue()
{
    int reponse = QMessageBox::question(this, "Interrogatoire",
    "Dites voir, je me posais la question comme ça, êtes-vous vraiment
    un Zéro ?", QMessageBox::Yes | QMessageBox::No);

    if (reponse == QMessageBox::Yes)
    {
        QMessageBox::information(this, "Interrogatoire", "Alors
        bienvenue chez les Zéros !");
    }
    else if (reponse == QMessageBox::No)
    {
        QMessageBox::critical(this, "Interrogatoire", "Tricheur !
        menteur ! Voleur ! Ingrat ! Lâche ! Traître !\nSors d'ici ou
        j'appelle la police !");
    }
}
```

Voici un schéma de ce qui peut se passer :



C'est ma foi clair, non ? 😊



Petite précision quand même : le type de retour exact de la méthode n'est pas `int` mais `QMessageBox::StandardButton`. Or, il s'agit là d'une énumération, et comme vous le savez probablement, une énumération n'est rien d'autre que le remplacement de nombres par des mots plus lisibles. Utiliser un `int` revient donc strictement au même. Si un rappel sur les énumérations s'impose parce que je viens de vous parler en chinois, relisez donc le [cours sur les énumérations](#) issu du tutoriel du langage C.

Saisir une information

Les boîtes de dialogues précédentes étaient un peu limitées car, à part présenter différents boutons, on ne pouvait pas trop interagir avec l'utilisateur.

Si vous souhaitez que votre utilisateur saisisse une information, ou encore fasse un choix parmi une liste, les boîtes de dialogue de saisie sont idéales. Elles sont gérées par la classe **QInputDialog**, que je vous conseille d'inclure dès maintenant dans `MaFenetre.h`.

Les boîtes de dialogue "saisir une information" peuvent être de 4 types. Nous allons les voir dans l'ordre :

- I. Saisir un texte
- II. Saisir un entier
- III. Saisir un nombre décimal (double)
- IV. Choisir un élément parmi une liste

Chacune de ces fonctionnalités est assurée par une méthode statique différente.

Saisir un texte (`QInputDialog::getText`)

La méthode statique `getText()` ouvre une boîte de dialogue qui permet à l'utilisateur de saisir un texte. Son prototype est :

Code : C++

```
QString QInputDialog::getText ( QWidget * parent, const QString &
title, const QString & label, QLineEdit::EchoMode mode =
QLineEdit::Normal, const QString & text = QString(), bool * ok = 0,
Qt::WindowFlags f = 0 );
```

Vous pouvez tout d'abord constater que la méthode retourne un `QString`, c'est-à-dire une chaîne de caractères de Qt. Les paramètres signifient, dans l'ordre :

- **parent** : pointeur vers la fenêtre parente. Peut être mis à `NULL` pour ne pas indiquer de fenêtre parente.
- **title** : titre de la fenêtre affiché en haut.
- **label** : texte affiché dans la fenêtre.
- **mode** : mode d'édition du texte. Permet de dire si on veut que les lettres s'affichent quand on tape, ou si elles doivent être remplacées par des astérisques (pour les mots de passe) ou si aucune lettre ne doit s'afficher. Toutes les options sont dans [la doc](#). Par défaut, les lettres s'affichent normalement (`QLineEdit::Normal`).
- **text** : le texte par défaut dans la zone de saisie.
- **ok** : un pointeur vers un booléen pour que Qt puisse vous dire si l'utilisateur a cliqué sur OK ou sur Annuler.
- **f** = quelques flags (options) permettant d'indiquer si la fenêtre est modale (bloquante) ou pas. Les valeurs possibles sont détaillées par [la doc](#).

Heureusement, comme vous pouvez le constater en lisant le prototype, certains paramètres possèdent des valeurs par défaut ce qui fait qu'ils ne sont pas obligatoires.

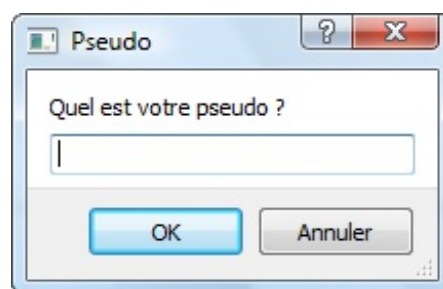
Reprenons notre code de tout à l'heure et cette fois, au lieu d'afficher une `QMessageBox`, nous allons afficher une `QInputDialog` lorsqu'on clique sur le bouton de la fenêtre.

Code : C++

```
void MaFenetre::ouvrirDialogue()
{
    QString pseudo = QInputDialog::getText(this, "Pseudo", "Quel est
votre pseudo ?");
}
```

En une ligne, je crée un `QString` et je lui affecte directement la valeur retournée par la méthode `getText()`. J'aurais aussi bien pu faire la même chose en deux lignes, mais ça aurait été plus long et je suis une feignasse 😊

La boîte de dialogue devrait ressembler à cela :



On peut aller plus loin et vérifier si le bouton OK a été actionné, et si c'est le cas on peut alors afficher le pseudo de l'utilisateur dans une `QMessageBox`.

Code : C++

```
void MaFenetre::ouvrirDialogue()
{
    bool ok = false;
    QString pseudo = QInputDialog::getText(this, "Pseudo", "Quel est
votre pseudo ?", QLineEdit::Normal, QString(), &ok);

    if (ok && !pseudo.isEmpty())
    {
        QMessageBox::information(this, "Pseudo", "Bonjour " + pseudo
+ ", ça va ?");
    }
}
```

```
    }  
    else  
    {  
        QMessageBox::critical(this, "Pseudo", "Vous n'avez pas voulu  
donner votre nom... sniff.");  
    }  
}
```

Ici, on crée un booléen qui va recevoir l'information "Le bouton OK a-t-il été cliqué ?".

Pour pouvoir l'utiliser dans la méthode `getText`, il faut donner tous les paramètres avant qu'on ne souhaite pourtant pas changer ! C'est un des défauts des paramètres par défaut en C++ : si le paramètre que vous voulez renseigner est tout à la fin (à droite), il faudra alors absolument renseigner tous les paramètres qui sont avant !

J'ai donc envoyé des valeurs par défaut aux paramètres qui étaient avant, à savoir **mode** et **text**.

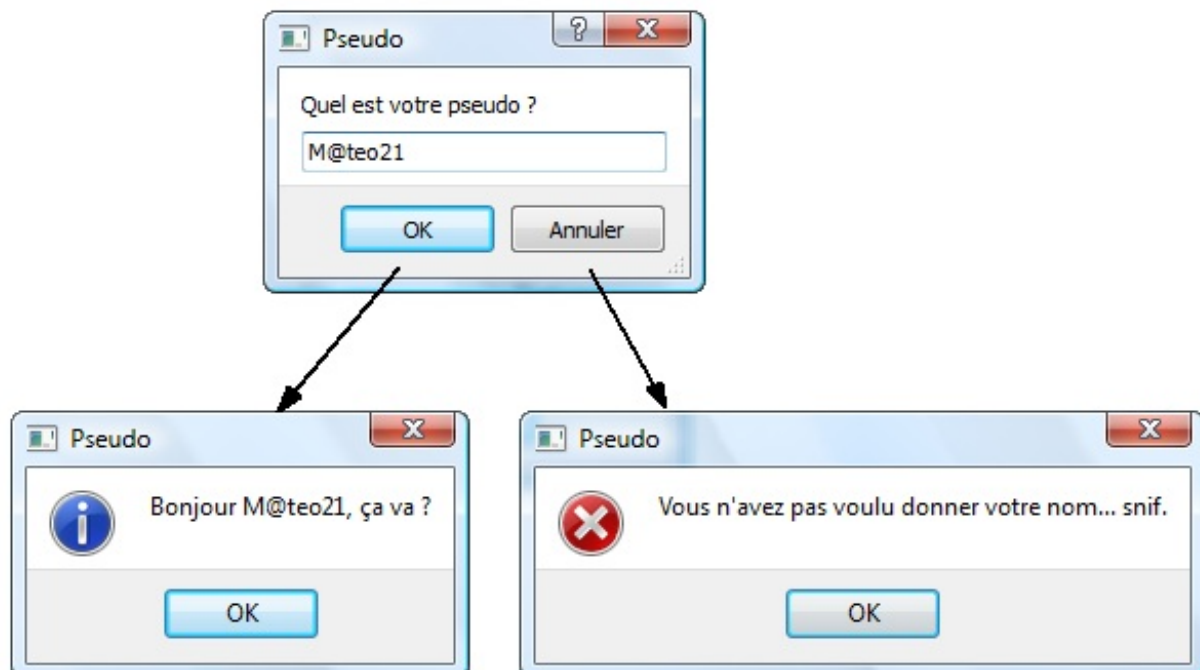
Comme j'ai donné un pointeur vers mon booléen à la méthode, celle-ci va le remplir pour indiquer si oui ou non le bouton a été cliqué.

Je peux ensuite faire un test, d'où la présence de mon `if`. Je vérifie 2 choses :

- Si le bouton OK a été cliqué
- Et si le texte n'est pas vide (la méthode `isEmpty` de `QString` sert à faire ça, vous ne pouviez pas la connaître, sauf en lisant la [doc de QString](#) bien sûr 😊).

Si un pseudo a été entré et que l'utilisateur a cliqué sur OK, alors une boîte de dialogue lui souhaite la bienvenue. Sinon, une erreur est affichée.

Ce schéma présente ce qui peut se produire :



Exercice : essayez d'afficher le pseudo de l'utilisateur quelque part sur la fenêtre mère, par exemple sur le bouton.

Saisir un entier (`QInputDialog::getInteger`)

La méthode `getInteger` devrait vous paraître simple maintenant que vous connaissez `getText`. Son prototype est :

Code : C++

```
int QInputDialog::getInteger ( QWidget * parent, const QString &
title, const QString & label, int value = 0, int minValue =
-2147483647, int maxValue = 2147483647, int step = 1, bool * ok = 0,
Qt::WindowFlags f = 0 );
```

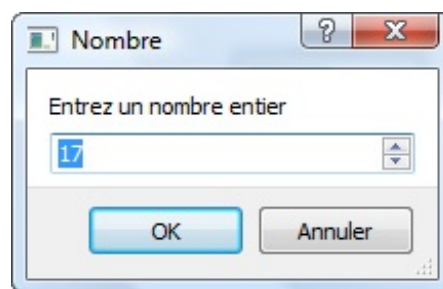
Elle retourne un int comme prévu.

Vous noterez les paramètres value (valeur par défaut), minValue (valeur minimale autorisée), maxValue (valeur maximale autorisée) et step, le pas d'incrémentement lorsqu'on clique sur les petites flèches (vous allez voir).

Testons ça avec les paramètres obligatoires, ça sera suffisant :

Code : C++

```
int entier = QInputDialog::getInteger(this, "Nombre", "Entrez un
nombre entier");
```



Les petites flèches à droite permettent à l'utilisateur d'incrémenter (ou de décrémenter) le nombre affiché. Le rôle du paramètre step est d'indiquer la valeur du pas d'incrémentement. Par défaut il est de 1.

Par exemple si je clique sur la flèche vers le haut alors que le nombre saisi est 12 et que j'ai mis un pas d'incrémentement de 10, le nombre deviendra 22.

Le nombre saisi est retourné par la méthode dans un entier, à vous de le traiter pour faire ce que bon vous semblera avec 😊

Saisir un nombre décimal (QInputDialog::getDouble)

La saisie d'un double est pratiquement identique à celle d'un entier, à la différence près qu'il y a un paramètre qui permet d'indiquer le nombre maximal de chiffres après la virgule autorisés (paramètre *decimals*).

Code : C++

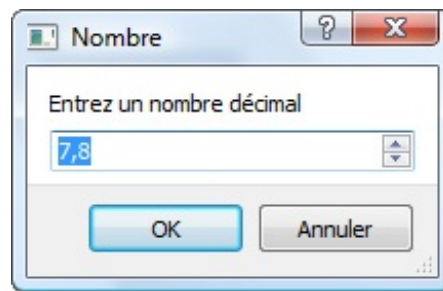
```
double QInputDialog::getDouble ( QWidget * parent, const QString &
title, const QString & label, double value = 0, double minValue =
-2147483647, double maxValue = 2147483647, int decimals = 1, bool *
ok = 0, Qt::WindowFlags f = 0 );
```

Petit test :

Code : C++

```
double nombreDecimal = QInputDialog::getDouble(this, "Nombre",
```

```
"Entrez un nombre décimal");
```



Choix d'un élément parmi une liste (QInputDialog::getItem)

Si l'utilisateur doit faire son choix dans une liste, cette méthode permet d'afficher les choix possibles dans une boîte de dialogue avec un menu déroulant.

Son prototype est :

Code : C++

```
QString QInputDialog::getItem ( QWidget * parent, const QString & title, const QString & label, const QStringList & list, int current = 0, bool editable = true, bool * ok = 0, Qt::WindowFlags f = 0 );
```

Il y a quelques nouveaux paramètres que je dois expliquer :

- **list** : la liste des choix possibles, envoyée via un objet de type QStringList (liste de chaînes) à construire au préalable.
- **current** : le numéro du choix qui doit être sélectionné par défaut.
- **editable** : un booléen qui indique si l'utilisateur a le droit d'entrer sa propre réponse (comme avec getText) ou s'il est obligé de faire un choix parmi la liste.

Toute la "difficulté", vous l'aurez compris, consiste à créer cette liste de choix. La doc nous dit qu'il faut envoyer un objet de type QStringList, allons donc voir la [doc de QStringList](#) !

Hmm...

Hmm hmm...

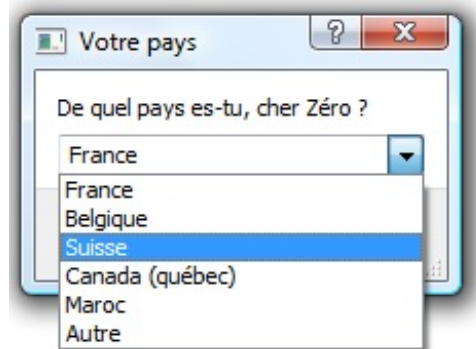
Intéressant. Bon le constructeur ne permet pas d'envoyer un nombre infini de chaînes à la liste, par contre on peut voir dans la doc que l'opérateur << est surchargé. Cela va nous permettre de "remplir" notre liste de chaînes très facilement !

Code : C++

```
void MaFenetre::ouvrirDialogue()
{
    QStringList pays;
    pays << "France" << "Belgique" << "Suisse" << "Canada (québec)"
    << "Maroc" << "Autre";
    QInputDialog::getItem(this, "Votre pays", "De quel pays es-tu, cher Zéro ?", pays);
}
```



Pensez à inclure le header de la classe `QStringList` avant de vous en servir, sinon le compilateur vous dira que la classe `QStringList` est indéfinie !



Et voilà, obstacle surmonté avec succès 😊

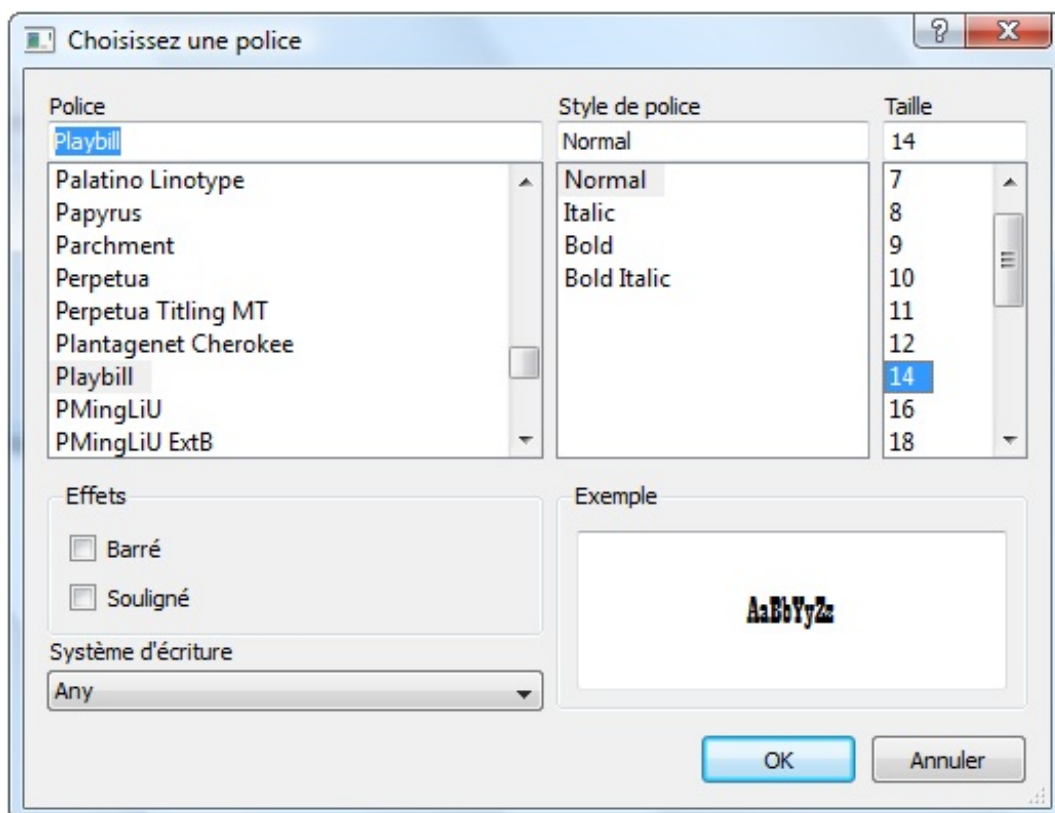


Si, pour une raison ou une autre, vous ne souhaitez pas utiliser l'opérateur surchargé `<<`, il existe des méthodes qui permettent d'ajouter des éléments un à un. Ces méthodes ne sont pas dans la classe `QStringList`, mais dans sa classe mère `QList`. On peut par exemple citer `append()` qui permet d'ajouter un élément à la fin de la liste.

Je dis ça pour vous rappeler de toujours regarder les méthodes de la classe mère si ce que vous cherchez n'est pas dans la liste des méthodes propres à votre classe.

Sélectionner une police

La boîte de dialogue "Sélectionner une police" est une des boîtes de dialogue standard les plus connues. Nul doute que vous l'avez déjà rencontrée dans l'un de vos programmes favoris.



La boîte de dialogue de sélection de police est gérée par la classe `QFontDialog`. Celle-ci propose en gros une seule méthode statique surchargée (il y a plusieurs façons de l'utiliser), comme vous pouvez le constater sur la [doc de QFontDialog](#).

Prenons le prototype le plus compliqué, juste pour la forme 😊

Code : C++

```
QFont getFont ( bool * ok, const QFont & initial, QWidget * parent,
const QString & caption )
```

Les paramètres se comprennent normalement assez facilement.

On retrouve notre pointeur vers un booléen "ok" qui permet de savoir si l'utilisateur a cliqué sur OK ou a annulé.

On peut spécifier une police par défaut (initial), il faudra envoyer un objet de type QFont. Voilà justement que la classe QFont réapparaît 😊

Enfin, la chaîne caption correspond au message qui sera affiché en haut de la fenêtre.

Enfin, et surtout, la méthode retourne un objet de type QFont correspondant à la police qui a été choisie.

Testons ! Histoire d'aller un peu plus loin, je propose que la police que nous aurons sélectionnée soit immédiatement appliquée au texte de notre bouton, par l'intermédiaire de la méthode setFont() que nous avons appris à utiliser il y a quelques chapitres.

Code : C++

```
void MaFenetre::ouvrirDialogue()
{
    bool ok = false;

    QFont police = QFontDialog::getFont(&ok, m_boutonDialogue-
    >font(), this, "Choisissez une police");

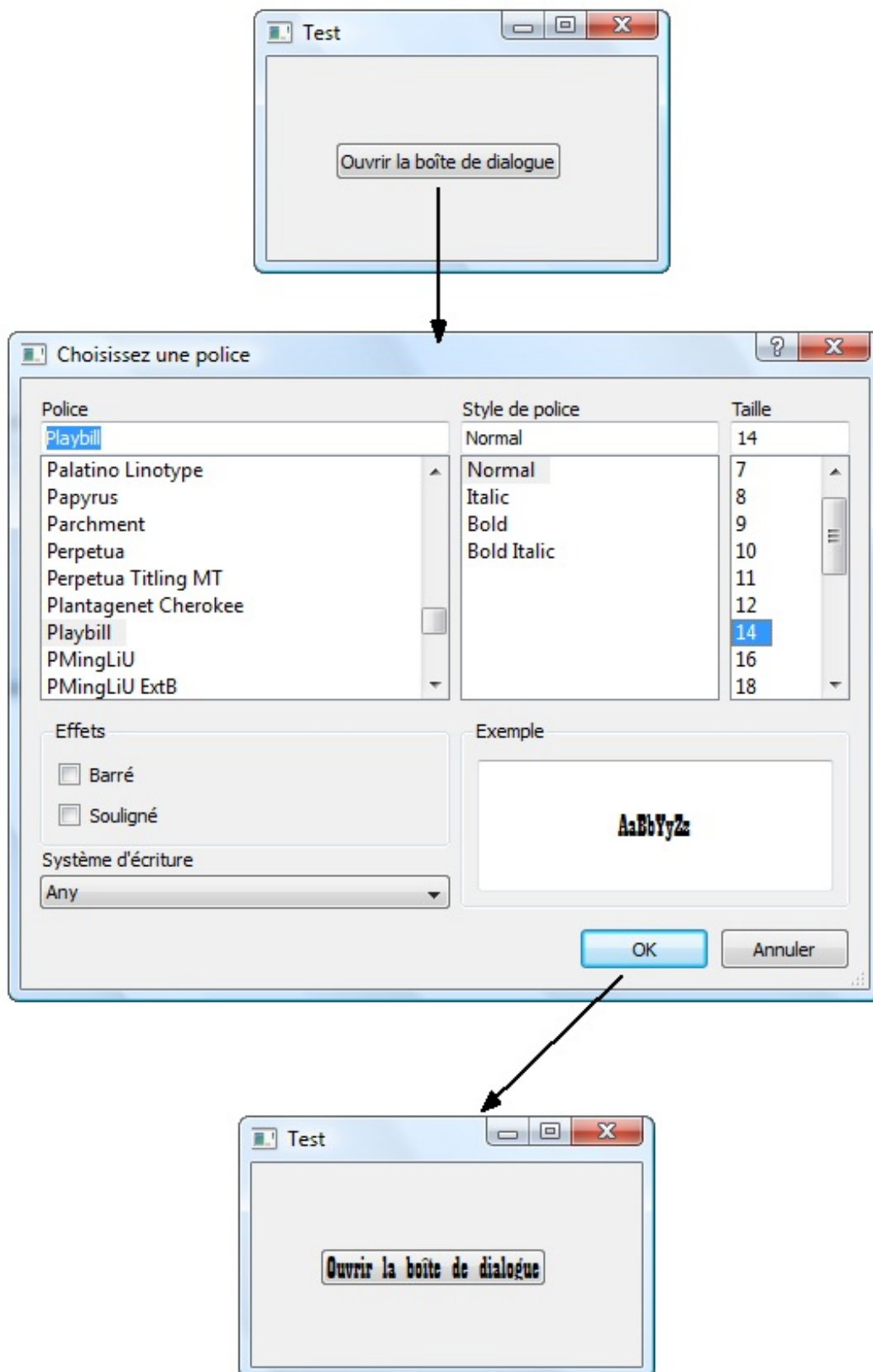
    if (ok)
    {
        m_boutonDialogue->setFont(police);
    }
}
```

La méthode getFont prend comme police par défaut celle qui est utilisée par notre bouton m_boutonDialogue (rappelez-vous, font() est une méthode accesseur qui renvoie un QFont).

On teste si l'utilisateur a bien validé la fenêtre, et si c'est le cas on applique la police qui vient d'être choisie à notre bouton.

C'est l'avantage de travailler avec les classes de Qt : elles sont cohérentes. La méthode getFont renvoie un QFont, et ce QFont nous pouvons l'envoyer à notre tour à notre bouton pour qu'il change d'apparence.

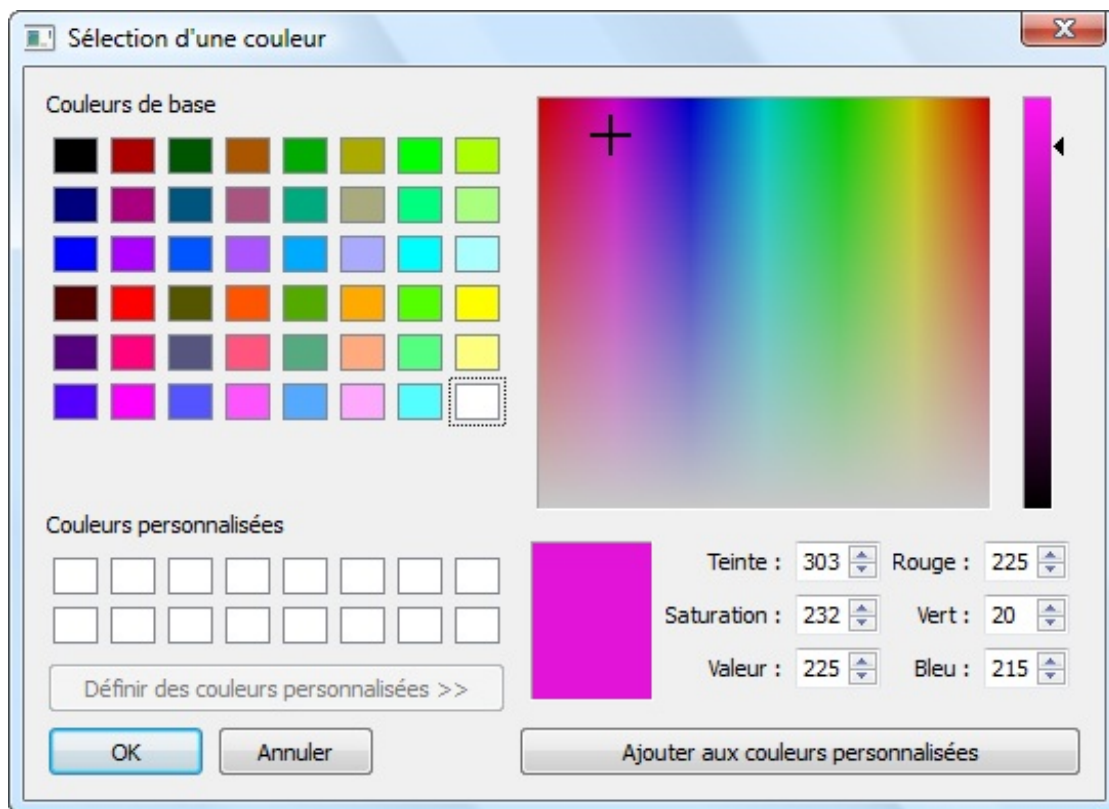
Le résultat ? Le voici :



Attention le bouton ne se redimensionne pas tout seul. Vous pouvez le rendre plus large de base si vous voulez, ou bien le redimensionner après le choix de la police.

Sélectionner une couleur

Dans la même veine que la sélection de police, on connaît probablement tous la boîte de dialogue "Sélection de couleur".



Utilisez la classe **QColorDialog** et sa méthode statique `getColor()`.

Code : C++

```
QColor QColorDialog::getColor ( const QColor & initial = Qt::white,
QWidget * parent = 0 );
```

Elle retourne un objet de type `QColor`. Vous pouvez préciser une couleur par défaut, en envoyant un objet de type `QColor` ou en utilisant une des [constantes prédéfinies de couleur](#). En l'absence de paramètre, c'est la couleur blanche qui sera sélectionnée comme nous l'indique le prototype.

Si on veut tester le résultat en appliquant la nouvelle couleur au bouton, c'est un petit peu compliqué. En effet, il n'existe pas de méthode `setColor` pour les widgets, mais une méthode `setPalette` qui sert à indiquer une palette de couleurs. Je vous laisse vous renseigner plus amplement si vous le désirez sur la [classe QPalette](#) qui est intéressante.

Le code que je vous propose ci-dessous ouvre une boîte de dialogue de sélection de couleur, puis crée une palette dont la couleur du texte correspond à la couleur qu'on vient de sélectionner, et applique enfin cette palette au bouton :

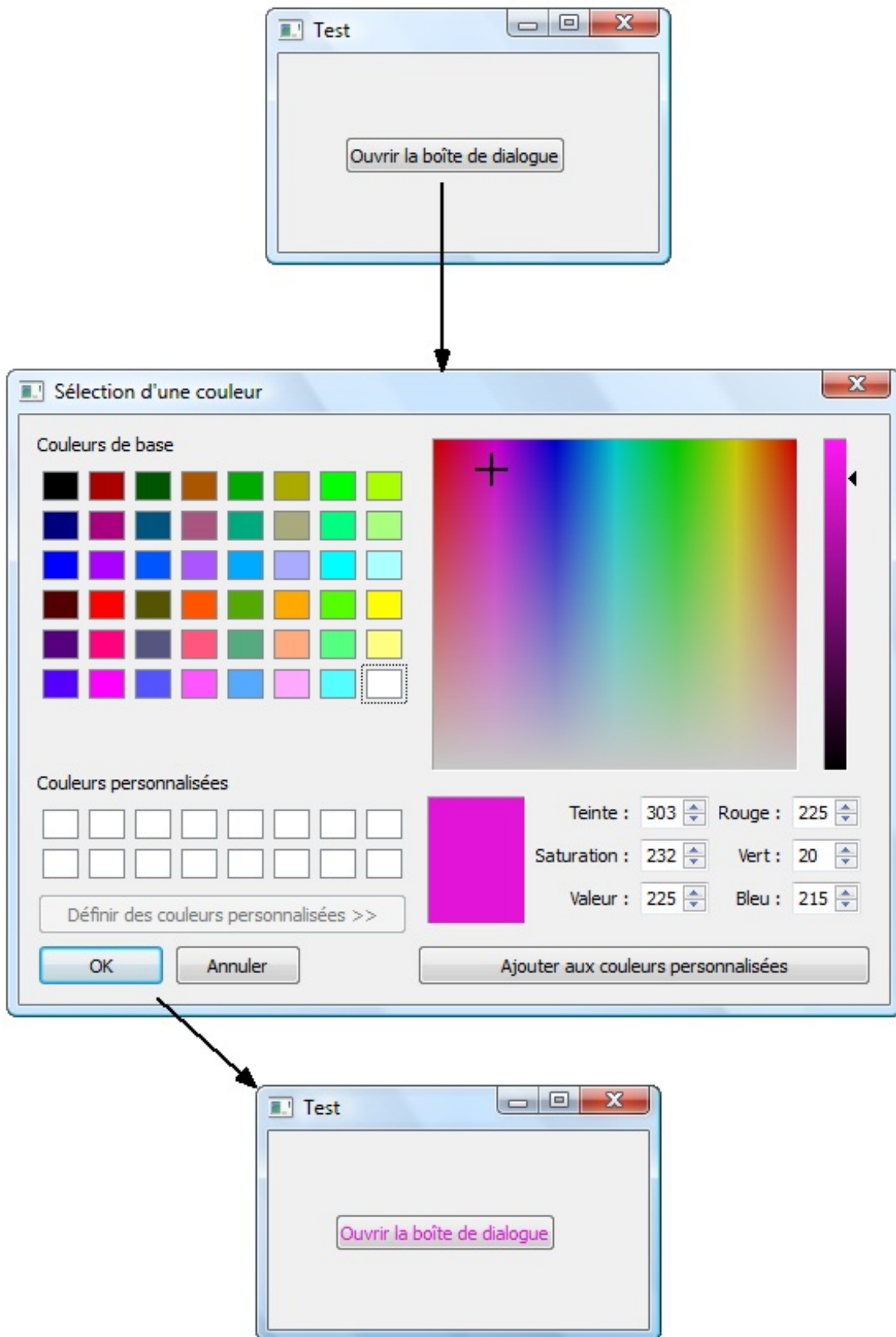
Code : C++

```
void MaFenetre::ouvrirDialogue()
{
    QColor couleur = QColorDialog::getColor(Qt::white, this);

    QPalette palette;
    palette.setColor(QPalette::ButtonText, couleur);
    m_boutonDialogue->setPalette(palette);
}
```

Je ne vous demande pas ici de comprendre comment fonctionne `QPalette`, qui est d'ailleurs une classe que je ne détaillerai pas plus dans le cours. A vous de vous renseigner sur elle si elle vous intéresse.

Le résultat de l'application est le suivant :



Sélection d'un fichier ou d'un dossier

Allez, plus que la sélection de fichiers et de dossiers et on aura fait le tour d'à peu près toutes les boîtes de dialogue usuelles qui existent ! 😊

La sélection de fichiers et de dossiers est gérée par la [classe QFileDialog](#) qui propose elle aussi des méthodes statiques faciles à utiliser.

Cette section sera divisée en 3 parties :

- Sélection d'un dossier existant
- Ouverture d'un fichier
- Enregistrement d'un fichier

Sélection d'un dossier existant (QFileDialog::getExistingDirectory)

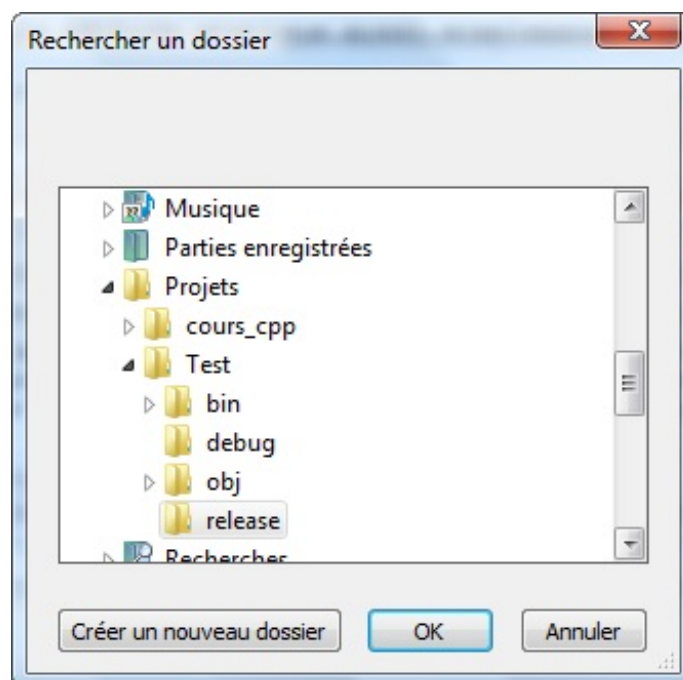
Bon je ne vous donne plus le prototype, vous devriez être assez grands pour le retrouver dans la doc 😊

On peut utiliser la méthode statique aussi simplement que comme ceci :

Code : C++

```
QString dossier = QFileDialog::getExistingDirectory(this);
```

Elle retourne un QString contenant le chemin complet vers le dossier demandé.
La fenêtre qui s'ouvre devrait ressembler à cela :



Ouverture d'un fichier (QFileDialog::getOpenFileName)

La célèbre boîte de dialogue "Ouverture d'un fichier" est gérée par `getOpenFileName()`.
Sans paramètres particuliers, la boîte de dialogue permet d'ouvrir n'importe quel fichier.

Vous pouvez néanmoins créer un filtre (4ème paramètre) pour afficher par exemple uniquement les images.

Ce code demande d'ouvrir un fichier image. Le chemin vers le fichier est stocké dans un QString, que l'on affiche ensuite via une QMessageBox :

Code : C++

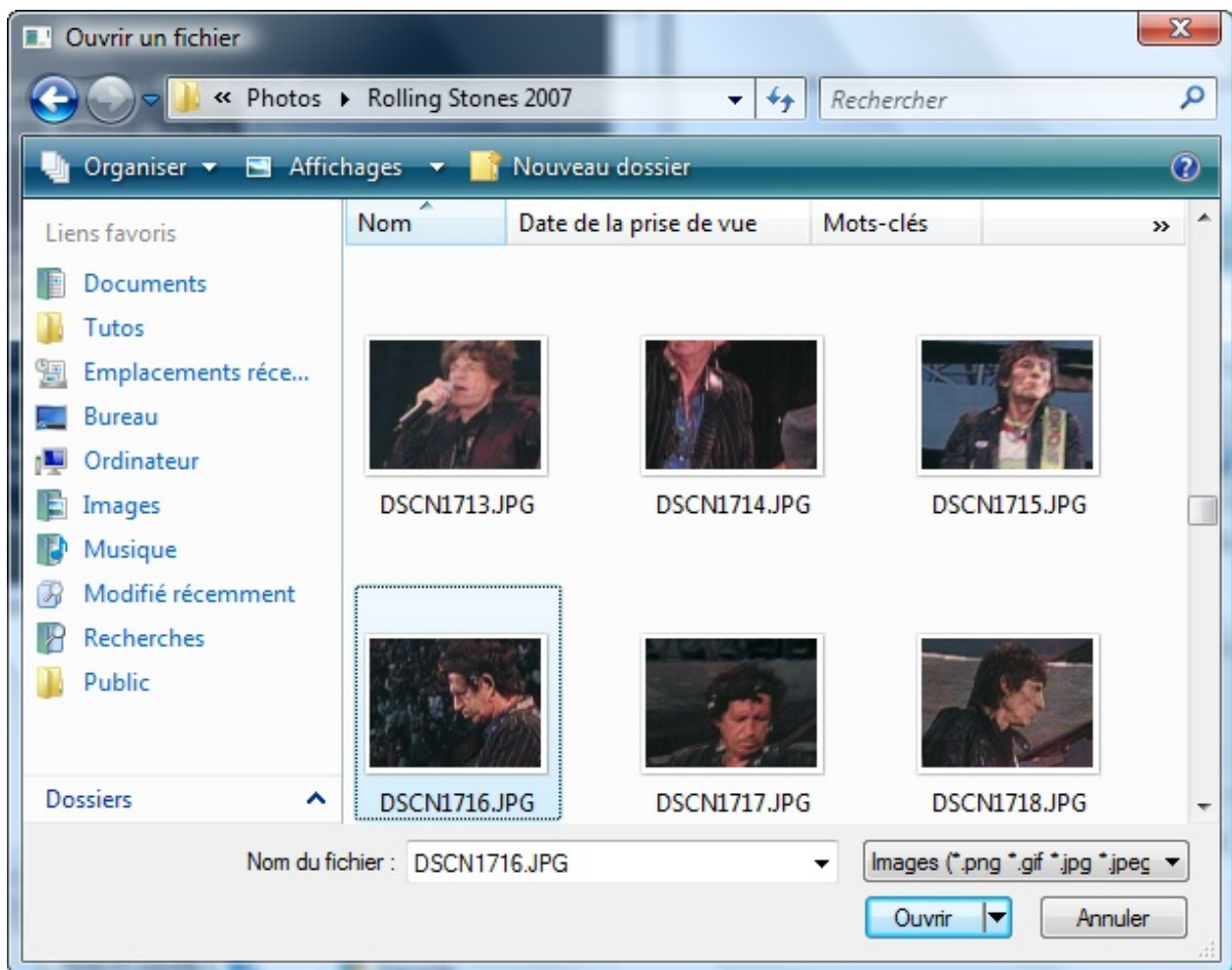
```
void MaFenetre::ouvrirDialogue()  
{
```

```
QString fichier = QFileDialog::getOpenFileName(this, "Ouvrir un  
fichier", QString(), "Images (*.png *.gif *.jpg *.jpeg)");  
QMessageBox::information(this, "Fichier", "Vous avez sélectionné  
:\n" + fichier);  
}
```

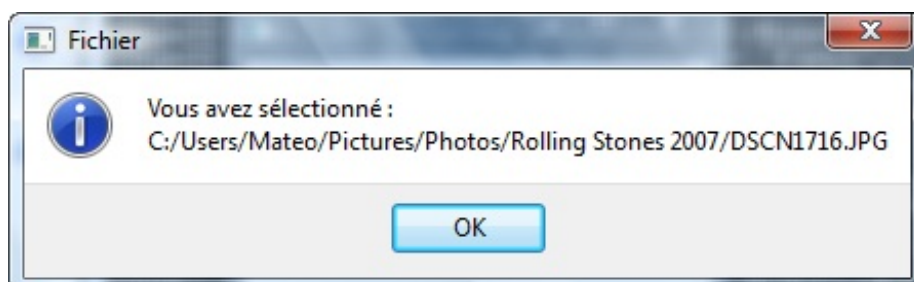
Le troisième paramètre de `getOpenFileName` est le nom du répertoire par défaut dans lequel l'utilisateur est placé. J'ai laissé la valeur par défaut (`QString()`, ce qui est équivalent à écrire `""`), donc la boîte de dialogue affichera par défaut le répertoire dans lequel est situé le programme.

Grâce au 4ème paramètre j'ai choisi de filtrer les fichiers. Seules les images de type PNG, GIF, JPG et JPEG s'afficheront.

Résultat :



La fenêtre bénéficie de toutes les options que propose votre OS, dont l'affichage des images sous forme de miniatures. Lorsque vous cliquez sur "Ouvrir", le chemin est enregistré dans un `QString` qui s'affiche ensuite dans une boîte de dialogue :



Le principe de cette boîte de dialogue est de vous donner le chemin complet vers le fichier, mais pas de vous ouvrir ce



fichier. C'est à vous ensuite de faire les opérations nécessaires pour ouvrir le fichier et l'afficher dans votre programme.

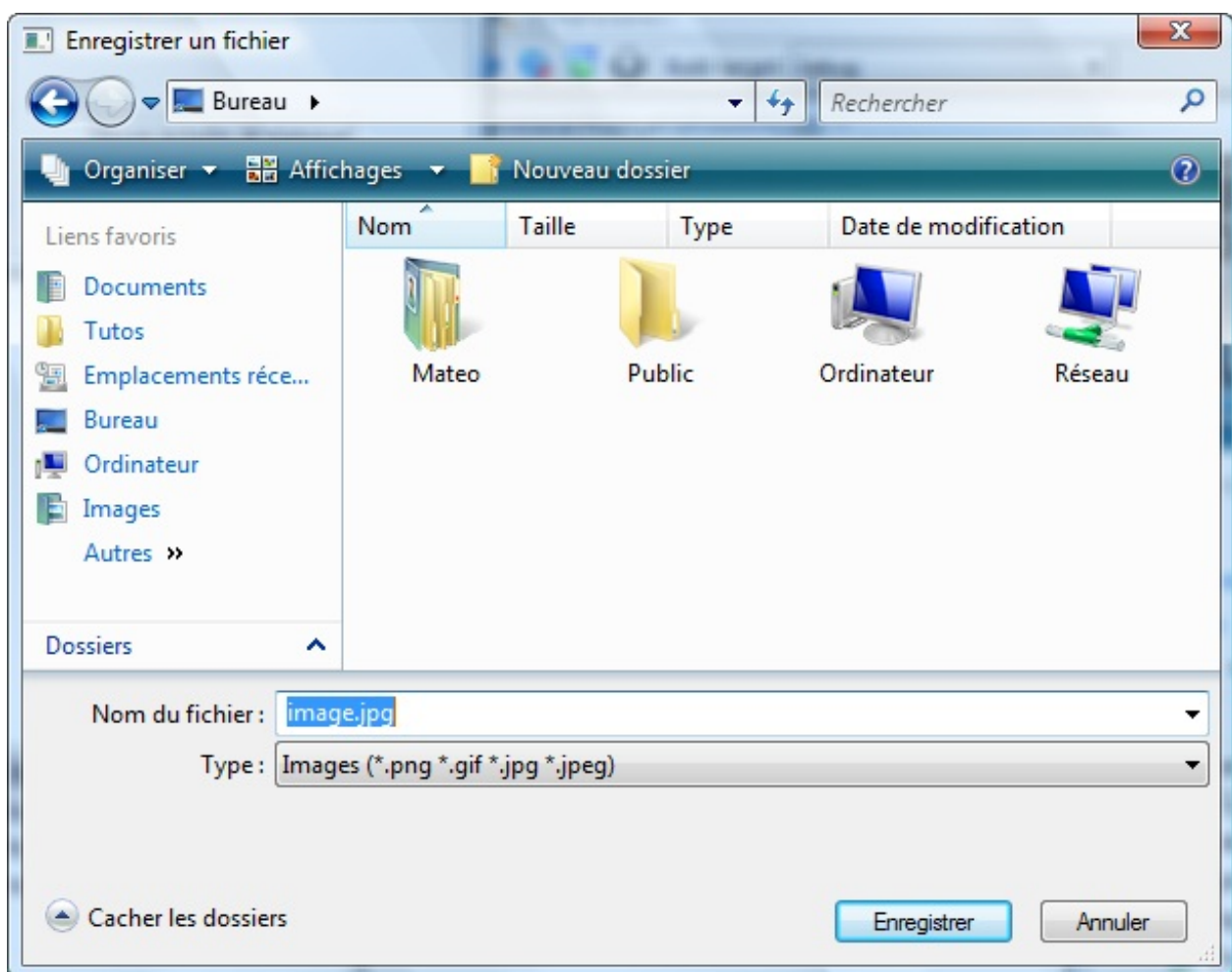
A noter aussi la fonction `getOpenFileNames` (notez le "s" à la fin) qui autorise la sélection de plusieurs fichiers. La principale différence est qu'au lieu de retourner un `QString`, elle retourne un `QStringList` (liste de chaînes). Tiens, comme on se retrouve !

Enregistrement d'un fichier (`QFileDialog::getSaveFileName`)

C'est le même principe que la méthode précédente, à la différence près que la personne peut cette fois spécifier un nom de fichier qui n'existe pas pour l'enregistrement. Le bouton "Ouvrir" est remplacé par "Enregistrer".

Code : C++

```
QString fichier = QFileDialog::getSaveFileName(this, "Enregistrer un  
fichier", QString(), "Images (*.png *.gif *.jpg *.jpeg)");
```



Je vous avais promis un chapitre simple, vous avez eu un chapitre simple !

En effet, les méthodes statiques ne sont rien d'autre que des "fonctions" comme en langage C, elles ne nécessitent donc pas de créer d'objets. Comme quoi, parfois le modèle objet est inadapté et ici c'était clairement le cas. Pour la plupart des classes que nous avons vues, on peut s'en sortir sans créer le moindre objet.

Ces considérations mises à part, le modèle objet reste quoiqu'il en soit très pratique lorsqu'on crée des GUI comme on le fait là. Et je peux vous dire qu'on n'a pas fini de tout découvrir 😊

A titre informatif, il existe quelques autres boîtes de dialogue usuelles un peu plus rares et surtout un peu plus complexes à utiliser. Je pense notamment à :

- **QProgressDialog** : affiche une boîte de dialogue avec une barre de progression et un bouton "Annuler". Cela permet de faire patienter l'utilisateur le temps qu'une longue opération s'exécute. Cette classe est très intéressante mais il vaut mieux qu'on la voie en pratique si on a l'occasion, car Qt cherche à estimer le temps restant pour savoir s'il doit afficher ou non la fenêtre. C'est plus intéressant de le voir dans un cas très concret donc.
- **QWizard** : affiche un assistant, avec les boutons "Suivant", "Précédent", "Terminer" ... Là encore il vaut mieux avoir un projet concret pour apprendre à utiliser cette classe car elle est assez complexe.

Ceci étant, vous pouvez aussi lire la documentation si vous en avez besoin maintenant, il y a tout ce qu'il faut dessus.



Mais... mais... je sais pas lire une doc moi, je sais pas où chercher l'information dont j'ai besoin, je suis perdu j'y comprends rien 🤔

Ah ouais ? C'est ce qu'on va voir !

On vous a pas encore fait de tuto pour vous apprendre à lire une doc à ce que je sache ? Alors c'est le moment d'apprendre !

Apprendre à lire la documentation de Qt

Voilà le chapitre le plus important de toute la partie sur Qt : celui qui va vous apprendre à **lire la documentation de Qt**.



Pourquoi est-ce que c'est si important de savoir lire la documentation ?

Parce que la documentation, c'est la bible du programmeur. Elle explique toutes les possibilités d'un langage ou d'une bibliothèque.

La documentation de Qt contient la liste des fonctionnalités de Qt. Toute la liste.

La documentation, c'est donc ce qu'il y a de plus complet mais... ça n'a rien à voir avec un tutoriel du Site du Zéro.

Déjà, il faudra vous y faire : la doc n'est disponible qu'en anglais (c'est valable pour Qt et pour la quasi-totalité des autres docs). Il faudra donc faire l'effort de lire de l'anglais, même si vous y êtes allergiques. En programmation, on peut rarement s'en sortir si on ne lit pas un minimum d'anglais technique.

D'autre part, la documentation est construite de manière assez déroutante quand on débute. Il faut être capable de "lire" et naviguer dans une documentation.

C'est précisément ce que ce chapitre va vous apprendre à faire 😊

Où trouver la doc ?

On vous dit que Qt propose une superbe documentation très complète qui vous explique tout son fonctionnement.

Oui, mais où peut-on trouver cette documentation au juste ? 🤔

Il y a en fait 2 moyens d'accéder à la doc :

- si vous avez internet : vous pouvez aller sur le **site de Nokia** (l'entreprise qui édite Qt) ;
- si vous n'avez pas internet : vous pouvez utiliser le programme **Qt Assistant** qui contient toute la doc. Vous pouvez aussi appuyer sur la touche F1 dans Qt Creator pour obtenir plus d'informations à propos du mot-clé sur lequel se trouve le curseur.

Avec internet : sur le site de Nokia

Personnellement, si j'ai accès à internet, j'ai tendance à préférer utiliser cette méthode pour lire la documentation. Il suffit d'aller sur le site web de Nokia, section documentation. L'adresse est simple à retenir :

<http://doc.qt.nokia.com>



Je vous conseille très fortement d'ajouter ce site dans vos **favoris**, et de faire en sorte qu'il soit visible !

Si vous ne faites pas un raccourci visible vers la doc, vous serez moins tentés d'y aller... or le but c'est justement que vous preniez le réflexe d'y aller 😊

Un des principaux avantages à aller chercher la doc sur internet, c'est que l'on est assuré d'avoir la doc la plus à jour. En effet, s'il y a des nouveautés ou des erreurs, on est certain en allant sur le net d'en avoir la dernière version.

Lorsque vous arrivez sur la doc, la page suivante s'affiche :



Online Reference Documentation

Qt	Tools	Addons
C++ Application Development Framework	Tools for Qt Development	Components and Addons for Qt
<ul style="list-style-type: none"> ■ Qt 4.7 Latest Qt 4.7 snapshot ■ Qt 4.6 / Qt Embedded 4.6 ■ Qt 4.5 / Qt Embedded 4.5 ■ Qt 4.4 / Qt Embedded 4.4 ■ Qt 4.3 / Qt Embedded 4.3 ■ Qt 4.2 / Qt Embedded 4.2 ■ Qt 4.1 / Qt Embedded 4.1 ■ Qt 4.0 ■ Qt 3.3 · Platforms · Compilers 	Nokia Qt SDK <ul style="list-style-type: none"> ■ Nokia Qt SDK 1.1 ■ Nokia Qt SDK 1.0.1 ■ Nokia Qt SDK 1.0 Qt Creator: <ul style="list-style-type: none"> ■ Latest Qt Creator snapshot ■ Qt Creator 2.1 ■ Qt Creator 2.0.1 ■ Qt Creator 2.0 ■ Qt Creator 1.3 	Qt Solutions: <ul style="list-style-type: none"> ■ Solutions for Qt 4 ■ Solutions for Qt 3 Teambuilder <ul style="list-style-type: none"> ■ Teambuilder 1.3 ■ Teambuilder 1.2 ■ Teambuilder 1.0 Qt Mobility Project <ul style="list-style-type: none"> ■ Version 1.1 ■ Version 1.0

C'est la liste des produits liés à Qt. Nous nous intéressons au premier cadre en haut à gauche intitulé Qt. Vous pouvez voir la liste des différentes versions de Qt, depuis Qt 2.3.

Sélectionnez la version de Qt qui correspond à celle que vous avez installée (normalement la toute dernière).

Voici la page qui devrait s'afficher maintenant :

Qt HOME DEV LABS **DOC** BLOG

Qt Reference Documentation

[Qt 4.7](#)
[ALL VERSIONS](#)

Search index:

API Lookup

- [Class index](#)
- [Function index](#)
- [Modules](#)
- [Namespaces](#)
- [Global Declarations](#)
- [QML elements](#)

Qt Topics

- [Programming with Qt](#)
- [Device UIs & Qt Quick](#)
- [UI Design with Qt](#)
- [Cross-platform and Platform-specific](#)
- [Platform-specific info](#)
- [Qt and Key Technologies](#)
- [How-To's and Best](#)

Home
A A A |

Qt Developer Guide

Qt is a cross-platform application and UI framework. Using Qt, you can write web-enabled applications once and deploy them across desktop, mobile and embedded operating systems without rewriting the source code.

- [Getting started](#)
- [Installation](#)
- [How to learn Qt](#)
- [Tutorials](#)
- [Examples](#)
- [What's new in Qt 4.7](#)

Qt API

- [All Classes](#)
- [Programming](#)
- [Qt Quick](#)

C'est l'accueil de la doc pour votre version de Qt.



Si vous le voulez, vous pouvez mettre directement cette page en favoris, car tant que vous n'installez pas une nouvelle version de Qt sur votre PC, il est inutile d'aller lire les docs des autres versions.

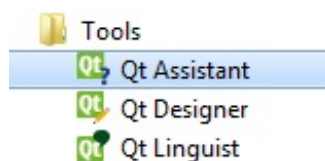
Nous allons détailler les différentes sections de cette page.

Mais avant... voyons voir comment accéder à la doc quand on n'a pas internet !

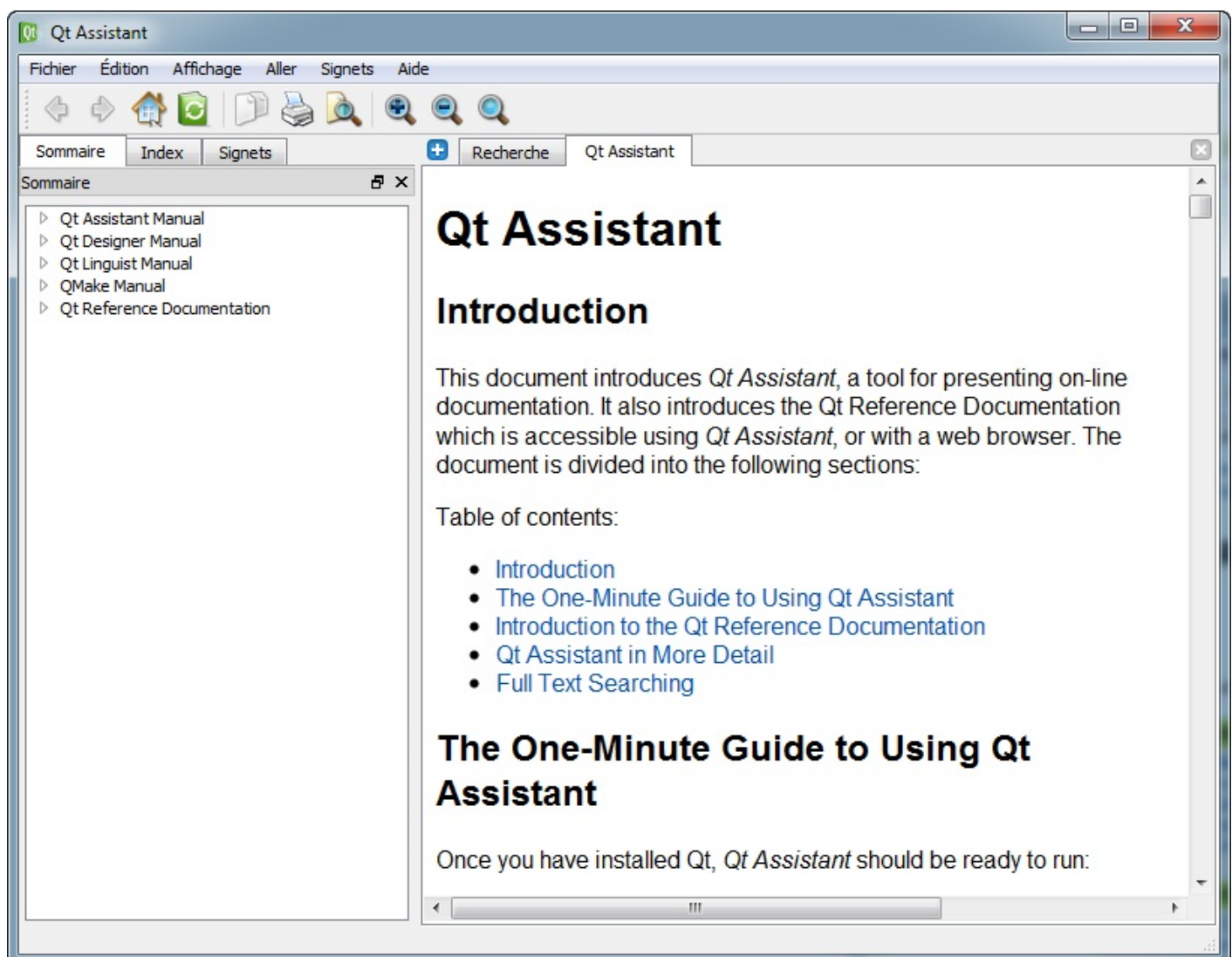
Sans internet : avec Qt Assistant

Si vous n'avez pas internet, pas de panique !

Qt a installé toute la documentation sur votre disque dur. Vous pouvez y accéder grâce au programme "Assistant" que vous retrouverez par exemple dans le menu Démarrer :



Qt Assistant se présente sous la forme d'un mini-navigateur qui contient la documentation de Qt :



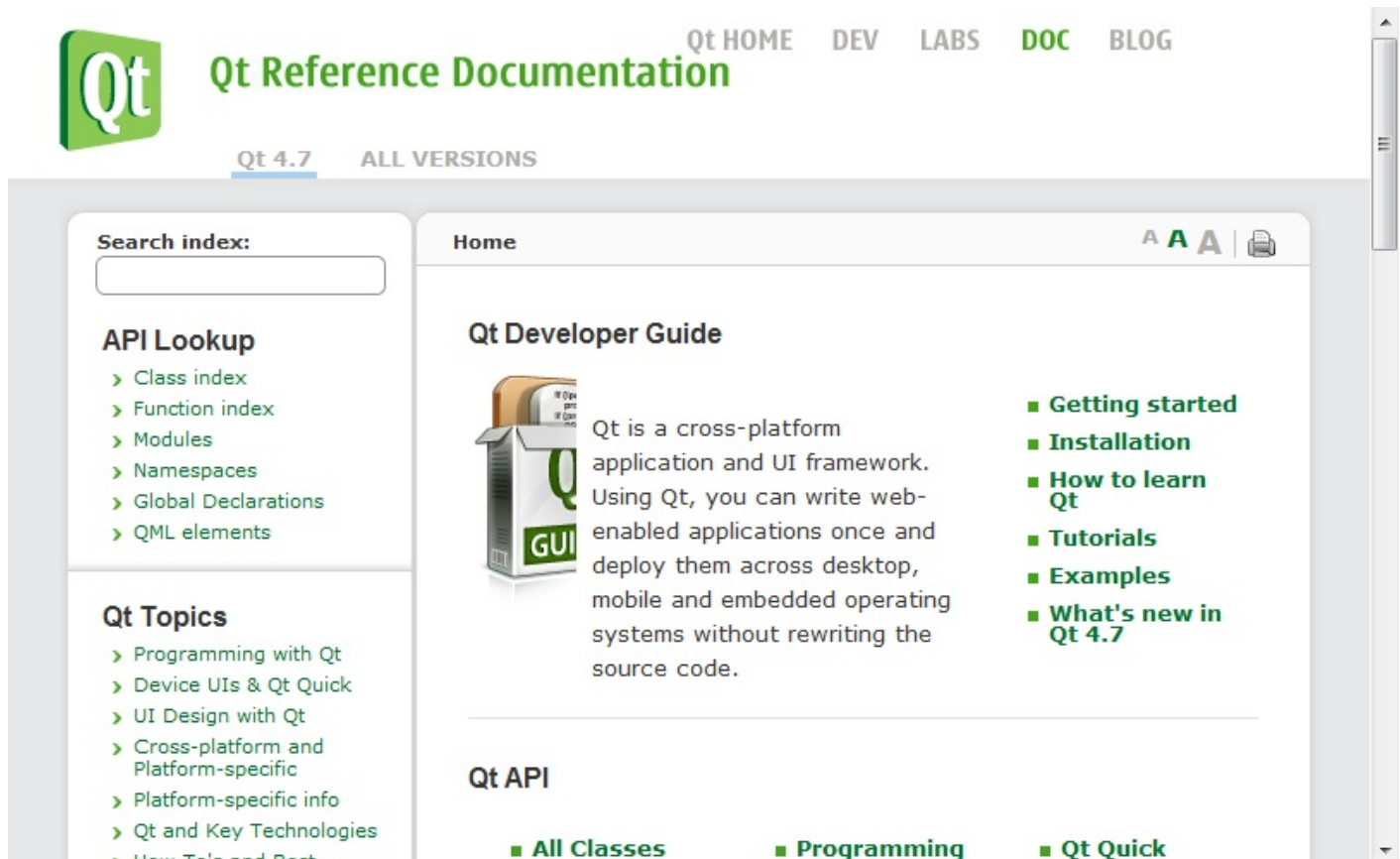


Vous ne disposez que de la documentation de Qt correspondant à la version que vous avez installée (c'est logique dans un sens). Si vous voulez lire la documentation d'anciennes versions de Qt (ou de futures versions en cours de développement) il faut obligatoirement aller sur internet.

Le logiciel Qt Assistant vous permet d'ouvrir plusieurs onglets différents en cliquant sur le bouton "+". Vous pouvez aussi effectuer une recherche grâce au menu à gauche de l'écran et rajouter des pages en favoris.

Les différentes sections de la doc

Lorsque vous arrivez à l'accueil de la doc, la page suivante s'affiche comme nous l'avons vu :



C'est le sommaire de la doc. Il vous suffit de naviguer à travers le menu de gauche. Celui-ci est découpé en 3 sections :

- API Lookup
- Qt Topics
- Examples

Les deux qui vont nous intéresser le plus sont API Lookup et Qt Topics. Vous pouvez regarder la section "Examples" aussi si vous le désirez, elle propose des exemples détaillés de programmes réalisés avec Qt.

Voyons ensemble ces 2 premières sections...

API Lookup

Cette section décrit dans le détail toutes les fonctionnalités de Qt. Ce n'est pas la plus lisible pour un débutant, mais c'est pourtant là qu'est le coeur de la doc :

- **Class index** : la liste de toutes les classes proposées par Qt. Il y en a beaucoup ! C'est une des sections que vous consulterez le plus souvent.
- **Function index** : c'est la liste de toutes les fonctions de Qt. Certaines de ces fonctions sont globales, d'autres sont des

fonctions membres (c'est-à-dire des méthodes de classe !). C'est une bonne façon pour vous d'accéder à la description d'une fonction.

- **Modules** : très intéressante, cette section répertorie les classes de Qt en fonction des modules. Qt étant découpé en plusieurs modules (Qt Core, Qt GUI...), cela vous permet de voir un peu comment est architecturé Qt globalement. Je vous invite à jeter un coup d'oeil en premier à cette section, c'est celle qui vous donnera le meilleur recul.
- **Namespaces** : la liste des espaces de noms employés par Qt pour ranger les noms de classes et de fonctions. Nous n'en aurons pas vraiment besoin.
- **Global Declarations** : toutes les déclarations globales de Qt. Ce sont des éléments qui sont accessibles partout dans tous les programmes Qt. Nous n'avons pas vraiment besoin d'étudier cela dans le détail.
- **QML elements** : une liste des éléments du langage QML de Qt, basé sur XML. Nous n'utiliserons pas QML dans ce cours, mais sachez que QML permet de créer des fenêtres d'une façon assez souple, basée sur le langage XML (qui ressemble au HTML).

Ceci étant, l'élément que vous utiliserez vraiment le plus souvent est le **champ de recherche** en haut du menu. Lorsque vous aurez une question sur le fonctionnement d'une classe ou d'une méthode, il vous suffira de rentrer son nom dans le champ de recherche pour être redirigé immédiatement vers la page qui présente son fonctionnement.

Qt Topics

Ce sont des pages de guide qui servent non seulement d'introduction à Qt, mais aussi de conseils pour ceux qui veulent utiliser Qt le mieux possible (notamment la section "Best practices"). Bien sûr, tout est en anglais. 😊

La lecture de cette section peut être très intéressante et enrichissante pour vous. Vous n'avez pas besoin de la lire dans l'immédiat, car mon cours va vous permettre d'avoir une bonne introduction globale à Qt... mais si plus tard vous souhaitez aller plus loin, vous trouverez des articles très utiles dans cette section !

Comprendre la documentation d'une classe

Voilà la section la plus importante et la plus intéressante de ce chapitre : nous allons étudier la documentation d'une classe de Qt au hasard.

Chaque classe possède sa propre page, plus ou moins longue selon la complexité de la classe. Vous pouvez donc retrouver tout ce dont vous avez besoin de savoir sur une classe en lisant une seule page.

Bon, j'ai dit qu'on allait prendre une classe de Qt au hasard. Alors, voyons voir... sur qui ça va tomber... ah ! Je sais :

QLineEdit



Lorsque vous connaissez le nom de la classe et que vous voulez lire sa documentation, utilisez le champ de recherche en haut du menu. Vous pouvez aussi passer par le lien "All Classes" depuis le sommaire.

Vous devriez avoir une longue page qui s'affiche sous vos yeux ébahis, et qui commence par quelque chose comme ça :

The screenshot shows the Qt Reference Documentation page for the `QLineEdit` class. The page layout includes a sidebar on the left with sections: Search index, API Lookup (Class index, Function index, Modules, Namespaces, Global Declarations, QML elements), Qt Topics (Programming with Qt, Device UIs & Qt Quick, UI Design with Qt, Cross-platform and Platform-specific, Platform-specific info, Qt and Key Technologies, How-To's and Best Practices), and Examples (Examples, Tutorials, Demos, QML Examples). The main content area has a breadcrumb trail: Home > Modules > QtGui > QLineEdit. The title is "QLineEdit Class Reference". The description states: "The QLineEdit widget is a one-line text editor. [More...](#)". Below this is a code snippet: `#include <QLineEdit>`. It then says "Inherits QWidget." and lists two bullet points: "List of all members, including inherited members" and "Qt 3 support members". There is a "Public Types" section showing an enum: `enum EchoMode { Normal, NoEcho, Password, PasswordEchoOnEdit }`. The "Properties" section lists: `acceptableInput` (const bool), `alignment` (Qt::Alignment), `maxLength` (int), `modified` (bool), and `placeholderText` (QString). A "Contents" sidebar on the right lists: Public Types, Properties, Public Functions, Public Slots, Signals, Protected Functions, and Detailed Description.

Chaque documentation de classe suit exactement la même structure. Vous retrouverez donc les mêmes sections, les mêmes titres, etc.

Analysons à quoi correspond chacune de ces sections ! 😊

Introduction

Au tout début, vous pouvez lire une très courte introduction qui explique en quelques mots à quoi sert la classe.

Ici, nous avons : "*The QLineEdit widget is a one-line text editor.*", ce qui signifie, si vous avez bien révisé votre anglais, que ce widget est un éditeur de texte sur une ligne, comme le montre la capture d'écran ci-contre.

Le lien "[More...](#)" vous amène vers une description plus détaillée de la classe. En général, il s'agit d'un mini-tutoriel pour apprendre à utiliser la classe. Je vous recommande de toujours lire cette introduction quand vous travaillez avec une classe que vous ne connaissiez pas jusqu'alors.

Ça vous fera gagner beaucoup de temps car vous saurez "par où commencer" et "quelles sont les principales méthodes de la classe".

Ensuite, on vous donne le header à inclure pour pouvoir utiliser la classe dans votre code, en l'occurrence il s'agit de :

Code : C++

```
#include <QLineEdit>
```

Puis, vous avez une information très importante à côté de laquelle on passe souvent : la classe dont hérite votre classe. Ici, on voit que QWidget est le parent de QLineEdit. Donc QLineEdit récupère toutes les propriétés de QWidget. Ça a son importance comme nous allons le voir..

Voilà pour l'intro ! 😊

Maintenant, voyons voir les sections qui suivent...

Public Types

Les classes définissent parfois des types de données personnalisés, sous la forme de ce qu'on appelle des énumérations (j'en ai parlé dans mon cours de C pour ceux qui auraient un trou de mémoire !).

Ici, QLineEdit définit l'énumération EchoMode qui propose plusieurs valeurs : Normal, NoEcho, Password, etc.



Une énumération ne s'utilise pas "telle quelle". C'est juste une liste de valeurs, que vous pouvez renvoyer à une méthode spécifique qui en a besoin. Dans le cas de QLineEdit, c'est la méthode `setEchoMode(EchoMode)` qui en a besoin, car elle n'accepte que des données de type EchoMode..

Pour envoyer la valeur "Password", il faudra écrire : `setEchoMode(QLineEdit::Password)`.

Properties

Vous avez là toutes les propriétés d'une classe que vous pouvez lire et modifier.



Euh, ce ne sont pas des attributs ça par hasard ? 🤔

Si. Mais la doc ne vous affiche que les attributs pour lesquels Qt définit des accesseurs. Il y a de nombreux attributs "internes" à chaque classe que la doc ne vous montre pas car ils ne vous concernent pas.

Toutes les propriétés sont donc des attributs intéressants de la classe que vous pouvez lire et modifier. Comme je vous l'avais dit dans un chapitre précédent, Qt suit cette convention pour le nom des accesseurs :

- **propriete()** : c'est la méthode accesseur qui vous permet de lire la propriété ;
- **setPropriete()** : c'est la méthode accesseur qui vous permet de modifier la propriété.

Prenons par exemple la propriété text. C'est la propriété qui stocke le texte rentré par l'utilisateur dans le champ de texte QLineEdit.

Comme indiqué dans la doc, text est de type QString. Vous devez donc récupérer la valeur dans un QString.

Pour récupérer le texte entré par l'utilisateur dans une variable contenu, on fera donc :

Code : C++

```
QLineEdit monChamp("Contenu du champ");  
QString contenu = monChamp.text();
```

Pour modifier le texte présent dans le champ, on écrira :

Code : C++

```
QLineEdit monChamp;
monChamp.setText("Entrez votre nom ici");
```

Vous remarquerez que dans la doc, la propriété text est un lien. [Cliquez dessus](#). Cela vous amènera plus bas sur la même page vers une description de la propriété (que fait-elle ? à quoi sert-elle ?).

On vous y donne aussi le prototype des accesseurs :

- `QString text () const`
- `void setText (const QString &)`

Et enfin, parfois vous verrez comme là une mention "See also" (voir aussi) qui vous invite à aller voir d'autres propriétés ou méthodes de la classe qui ont un rapport avec celle que vous êtes en train de lire. Ici, on vous dit que les méthodes `insert()` et `clear()` pourraient vous intéresser. En effet, par exemple `clear()` vide le contenu du champ de texte, c'est donc une méthode intéressante en rapport avec la propriété qu'on était en train de lire.

TRES IMPORTANT : dans la liste des propriétés en haut de la page, notez les mentions "*56 properties inherited from QWidget*", et "*1 property inherited from QObject*". Comme `QLineEdit` hérite de `QWidget`, qui lui-même hérite de `QObject`, il possède du coup toutes les propriétés et toutes les méthodes de ses classes parentes !



En clair, les propriétés que vous voyez là ne sont qu'un tout petit bout des possibilités offertes par `QLineEdit`. Si vous cliquez sur le lien `QWidget`, on vous amène vers la [liste des propriétés de QWidget](#). Vous disposez aussi de toutes ces propriétés dans un QLineEdit !

Vous pouvez donc utiliser la propriété `width` (largeur) qui est définie dans `QWidget` pour modifier la largeur de votre `QLineEdit`. Toute la puissance de l'héritage est là ! Tous les widgets possèdent donc ces propriétés "de base", ils n'ont plus qu'à définir des propriétés qui leur sont spécifiques.

J'insiste bien dessus car au début je me disais souvent : "*Mais pourquoi il y a aussi peu de choses dans cette classe ?*". En fait, il ne faut pas s'y fier et toujours regarder les classes parentes dont hérite la classe qui vous intéresse. Tout ce que les classes parentes possèdent, vous y avez accès aussi.

Public Functions

C'est bien souvent la section la plus importante. Vous y trouverez toutes les méthodes publiques (parce que les privées ne vous concernent pas) de la classe. On trouve dans le lot :

- le (ou les) constructeur(s) de la classe. Très intéressant pour savoir comment créer un objet à partir de cette classe ;
- les accesseurs de la classe (comme `text()` et `setText()` qu'on vient de voir), basés sur les attributs ;
- et enfin d'autres méthodes publiques qui ne sont ni des constructeurs ni des accesseurs et qui effectuent diverses opérations sur l'objet. Par exemple : `home()`, qui ramène le curseur au début du champ de texte.

Cliquez sur le nom d'une méthode pour en savoir plus sur son rôle et son fonctionnement.

Lire et comprendre le prototype

A chaque fois, il faut que vous lisiez attentivement le prototype de la méthode, c'est très important ! Le prototype à lui seul vous donne une grosse quantité d'informations sur la méthode.

Prenons l'exemple du constructeur. On voit qu'on a 2 prototypes :

- `QLineEdit (QWidget * parent = 0)`
- `QLineEdit (const QString & contents, QWidget * parent = 0)`

Vous noterez que certains paramètres sont facultatifs.

Si vous cliquez sur un de ces constructeurs, par exemple le [second](#), on vous explique la signification de chacun de ces paramètres.

On apprend que `parent` est un pointeur vers le widget qui "contiendra" notre `QLineEdit` (par exemple une fenêtre), et que `contents` est le texte qui doit être écrit dans le `QLineEdit` par défaut.

Cela veut dire, si on prend en compte que le paramètre `parent` est facultatif, qu'on peut créer un objet de type `QLineEdit` de 4 façons différentes :

Code : C++

```
QLineEdit monChamp(); // Appel du premier constructeur
QLineEdit monChamp(fenetre); // Appel du premier constructeur
QLineEdit monChamp("Entrez un texte"); // Appel du second
constructeur
QLineEdit monChamp("Entrez un texte", fenetre); // Appel du second
constructeur
```

C'est fou tout ce qu'un prototype peut raconter hein ? 🤔

Quand la méthode attend un paramètre d'un type que vous ne connaissez pas...



Je viens de voir la méthode `setAlignment()`, mais elle demande un paramètre de type `Qt::Alignment`. Comment je lui donne ça moi, je connais pas les `Qt::Alignment` !

Pas de panique. Il vous arrivera très souvent de tomber sur une méthode qui attend un paramètre d'un type qui vous est inconnu. Par exemple, vous n'avez jamais entendu parler de `Qt::Alignment`. Qu'est-ce que c'est que ce type ?

La solution pour savoir *comment envoyer un paramètre de type `Qt::Alignment`* consiste à cliquer dans la doc sur le lien [Qt::Alignment](#) (eh oui, ce n'est pas un lien par hasard !).

Ce lien vous amènera vers une page qui vous explique ce qu'est le type `Qt::Alignment`.

Il peut y avoir 2 types différents :

- **Les énumérations** : `Qt::Alignment` en est une. Les énumérations sont très simples à utiliser, c'est une série de valeurs. Il suffit d'écrire la valeur que l'on veut, comme le donne la [documentation de Qt::Alignment](#), par exemple `Qt::AlignCenter`. La méthode pourra donc être appelée comme ceci :

Code : C++

```
monChamp.setAlignment(Qt::AlignCenter);
```

- **Les classes** : parfois, la méthode attend un objet issu d'une classe précise pour travailler. Là c'est un peu plus compliqué : il va falloir créer un objet de cette classe et l'envoyer à la méthode.

Prenons par exemple [setValidator](#), qui attend un pointeur vers un `QValidator`. La méthode `setValidator` vous dit qu'elle

permet de vérifier si l'utilisateur a rentré un texte valide, ce qui peut être utile si vous voulez vérifier que l'utilisateur a bien rentré un nombre entier et non pas "Bonjour ça va ?" quand vous lui demandez son âge...

Si vous cliquez sur le lien [QValidator](#), on vous emmène vers la page qui explique comment utiliser la classe QValidator. Lisez le texte d'introduction pour comprendre ce que cette classe est censée faire, puis regardez les constructeurs afin de savoir comment créer un objet de type QValidator.

Parfois, comme là, c'est même un peu plus délicat. QValidator est une classe abstraite (c'est ce que vous dit l'intro de sa doc), ce qui signifie qu'on ne peut pas créer d'objet de type QValidator et qu'il faut utiliser une de ses classes filles 😬

Au tout début, la page de la doc de QValidator vous dit "Inherited by QDoubleValidator, QIntValidator, and QRegExpValidator". Cela signifie que ces classes héritent de QValidator et que vous pouvez les utiliser aussi. En effet, une classe fille est compatible avec la classe mère, comme nous l'avons déjà vu dans le [chapitre sur l'héritage](#).

Nous, nous voulons autoriser uniquement la personne à rentrer un nombre entier, nous allons donc utiliser [QIntValidator](#). Il faut créer un objet de type QIntValidator. Regardez ses constructeurs et choisissez celui qui vous convient.

Au final (ouf !), pour utiliser setValidator, on peut faire comme ceci :

Code : C++

```
QValidator *validator = new QIntValidator(0, 150, this);
monChamp.setValidator(validator);
```

... pour s'assurer que la personne ne rentrera qu'un nombre compris entre 0 et 150 ans (ça laisse de la marge 😊).

La morale de l'histoire, c'est qu'il ne faut pas avoir peur d'aller lire la documentation d'une classe dont a besoin la classe sur laquelle vous travaillez, et même des fois là d'aller voir les classes filles.

Ça peut faire un peu peur au début, mais c'est une gymnastique de l'esprit à acquérir. N'hésitez donc pas à sauter de lien en lien dans la doc pour arriver enfin à envoyer à cette `QObject::connect` de méthode un objet du type qu'elle attend ! 😊

Public Slots

Les slots sont des méthodes comme les autres, à la différence près qu'on peut aussi les connecter à un signal comme on l'a vu dans le chapitre sur les signaux et les slots.

Notez que rien ne vous interdit d'appeler un slot directement, comme si c'était une méthode comme une autre.

Par exemple, le `slot undo()` annule la dernière opération de l'utilisateur.

Vous pouvez l'appeler comme une bête méthode :

Code : C++

```
monChamp.undo();
```

... mais la particularité du fait que `undo()` soit un slot, c'est que vous pouvez aussi le connecter à un autre widget. Par exemple, on peut imaginer un menu Edition / Annuler dont le signal "cliqué" sera connecté au slot "undo" du champ de texte 😊



Tous les slots offerts par `QLineEdit` ne sont pas dans cette liste. Je me permets de vous rappeler une fois de plus qu'il faut penser à regarder les mentions comme "19 public slots inherited from `QWidget`", qui vous invitent à aller voir les slots de `QWidget` auxquels vous avez aussi accès.

C'est ainsi que vous découvrirez que vous disposez du slot `hide()` qui permet de masquer votre `QLineEdit`.

Signals

C'est la liste des signaux que peut envoyer un QLineEdit.

Un signal est un évènement qui s'est produit et que l'on peut connecter à un slot (le slot pouvant appartenir à cet objet ou à un autre).

Par exemple, le signal `textChanged()` est émis à chaque fois que l'utilisateur modifie le texte à l'intérieur du QLineEdit. Si vous le voulez, vous pouvez connecter ce signal à un slot pour qu'une action soit effectuée à chaque fois que le texte est modifié.

Attention encore une fois à bien regarder les signaux hérités de QWidget et QObject, car ils appartiennent aussi à la classe QLineEdit. Je sais que je suis lourd à force de répéter ça, inutile de me le dire, je le fais exprès pour que ça rentre 😊

Protected Functions

Ce sont des méthodes protégées. Elles ne sont ni public, ni private, mais protected.

Comme on l'a vu dans le chapitre sur l'héritage, ce sont des méthodes privées (auxquelles vous ne pouvez pas accéder directement en tant qu'utilisateur de la classe) mais qui seront héritées et donc réutilisables si vous créez une classe basée sur QLineEdit.

Il est très fréquent d'hériter des classes de Qt, on l'a d'ailleurs déjà fait avec QWidget pour créer une fenêtre personnalisée. Si vous héritez de QLineEdit, sachez donc que vous disposerez aussi de ces méthodes.

Additional Inherited Members

Si des éléments hérités n'ont pas été listés jusqu'ici, on les retrouvera dans cette section à la fin.

Par exemple, la classe QLineEdit ne définit pas de méthodes statiques, mais elle en possède quelques-unes héritées de QWidget et QObject.

Je vous rappelle qu'une méthode statique est une méthode qui peut être appelée sans avoir eu à créer d'objet. C'est un peu comme une fonction.

Il n'y a rien de bien intéressant avec QLineEdit, mais sachez par exemple que la classe QString possède de nombreuses méthodes statiques, comme `number()` qui convertit le nombre donné en une chaîne de caractères de type QString.

Code : C++

```
QString maChaine = QString::number(12);
```

Une méthode statique s'appelle comme ceci : `NomDeLaClasse::nomDeLaMethode()`.

On a déjà vu tout ça dans les chapitres précédents, je ne fais ici que des rappels 😊

Detailed description

C'est une description détaillée du fonctionnement de la classe. On y accède notamment en cliquant sur le lien "More..." après la très courte introduction du début.

C'est une section très intéressante que je vous invite à lire la première fois que vous découvrez une classe, car elle vous permet de comprendre avec du recul comment la classe est censée fonctionner.

Ce chapitre était absolument nécessaire car je suis convaincu que vous ne pouvez pas passer à côté de la doc.

Toutes les informations dont vous avez besoin y sont, le tout est d'être capable de les retrouver et de les comprendre. C'est, je l'espère, ce que ce chapitre vous aura aidés à faire. Il s'agissait de faire une sorte de "guide" pour rassurer les débutants qui n'ont jamais vraiment touché à une documentation.

Le concept pour apprendre un langage ou une bibliothèque est donc le suivant :

1. d'abord on lit des tutoriels qui nous permettent de savoir comment débiter et dans quelle direction chercher ;
2. et ensuite on consulte la doc pour connaître le détail des fonctions et des classes.

Il n'existe pas de tutoriel qui vous apprendra tout de A à Z sur une bibliothèque comme Qt par exemple. Ca n'aurait pas de sens et ce serait complètement stupide de chercher à faire ça étant donné que cela représenterait un travail énorme qui deviendrait obsolète dès la prochaine mise à jour de Qt.

Le but de mon tutoriel n'est donc pas de "tout vous apprendre" mais de vous apprendre à apprendre. Bien sûr, je ne vous lâche pas dans la nature comme ça : j'ai encore beaucoup de choses à vous expliquer dans ce tutoriel. Mais pensez à lire la doc en parallèle de mes cours, et une fois que vous aurez fini de lire ma prose, ayez le réflexe de consulter la doc à chaque fois que vous en avez besoin.

C'est un vrai réflexe de programmeur 😊

Positionner ses widgets avec les layouts

Comme vous le savez, une fenêtre peut contenir toutes sortes de widgets : des boutons, des champs de texte, des cases à cocher...

Placer ces widgets sur la fenêtre est une science à part entière. Je veux dire par là qu'il faut vraiment y aller avec méthode, si on ne veut pas que la fenêtre ressemble rapidement à un champ de bataille 🤪

Comment bien placer les widgets sur la fenêtre ?

Comment gérer les redimensionnements de la fenêtre ?

Comment s'adapter automatiquement à toutes les résolutions d'écran ?

On distingue 2 techniques différentes pour positionner des widgets :

- **Le positionnement absolu** : c'est celui que nous avons vu jusqu'ici, avec l'appel à la méthode *setGeometry* (ou *move*)... Ce positionnement est très précis, car on place les widgets au pixel près, mais cela comporte un certain nombre de défauts comme nous allons le voir.
- **Le positionnement relatif** : c'est le plus flexible et c'est celui que je vous recommande d'utiliser autant que possible. Nous allons l'étudier dans ce chapitre.

Le positionnement absolu et ses défauts

Nous allons commencer par voir le code Qt de base que nous allons utiliser dans ce chapitre, puis nous ferons quelques rappels sur le positionnement absolu que vous avez déjà utilisé sans savoir exactement ce que c'était 🤪

Le code Qt de base

Dans les chapitres précédents, nous avons créé un projet Qt constitué de 3 fichiers :

- **main.cpp** : contenait le main qui se chargeait juste d'ouvrir la fenêtre principale.
- **MaFenetre.h** : contenait l'en-tête de notre classe MaFenetre qui héritait de QWidget.
- **MaFenetre.cpp** : contenait l'implémentation des méthodes de MaFenetre, notamment du constructeur.

C'est l'architecture que l'on utilisera dans la plupart de nos projets Qt.

Toutefois, pour ce chapitre nous n'avons pas besoin d'une architecture aussi complexe, et nous allons faire comme dans les tout premiers chapitres Qt : nous allons juste utiliser un main (1 seul fichier : main.cpp).

Voici le code de votre projet, sur lequel nous allons commencer :

Code : C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

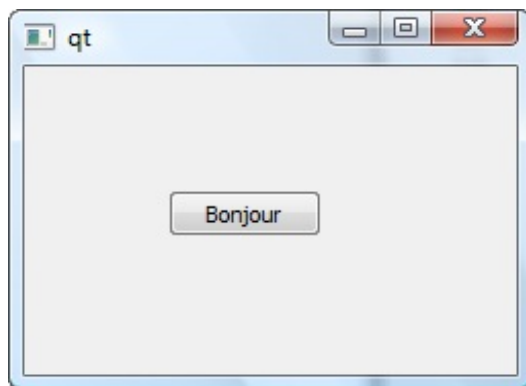
    QPushButton bouton("Bonjour", &fenetre);
    bouton.move(70, 60);

    fenetre.show();

    return app.exec();
}
```

C'est très simple : nous créons une fenêtre, et nous affichons un bouton que nous plaçons aux coordonnées (70, 60) sur la fenêtre.

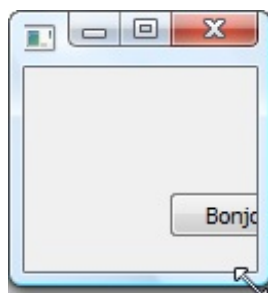
Le résultat est le suivant :



Les défauts du positionnement absolu

Dans le code précédent, nous avons positionné notre bouton de manière absolue en faisant `bouton.move(70, 60)` ; Le bouton a été très précisément placé 70 pixels sur la droite et 60 pixels plus bas.

Le problème... c'est que ce n'est pas flexible du tout. Imaginez que l'utilisateur s'amuse à redimensionner la fenêtre :



C'est moche, non ?

Le bouton ne bouge pas de place. Du coup, si on réduit la taille de la fenêtre, il sera coupé en deux, et pourra même disparaître si on réduit trop la taille.



Dans ce cas, pourquoi ne pas empêcher l'utilisateur de redimensionner la fenêtre ? On avait fait ça grâce à `setFixedSize` dans les chapitres précédents...

Oui, vous pouvez faire cela. C'est d'ailleurs ce que font le plus souvent les développeurs de logiciels qui positionnent leurs widgets en absolu. Cependant, l'utilisateur apprécie aussi de pouvoir redimensionner sa fenêtre. Ce n'est qu'une demi-solution.

D'ailleurs, il y a un autre problème que `setFixedSize` ne peut pas régler : le cas des résolutions d'écran plus petites que la vôtre. Imaginez que vous placiez un bouton 1200 pixels sur la droite parce que vous avez une grande résolution (1600 x 1200), et que l'utilisateur soit dans une résolution plus petite que vous (1024 x 768). Il ne pourra jamais voir le bouton, parce qu'il ne pourra jamais agrandir autant sa fenêtre !



Alors quoi ? Le positionnement absolu c'est mal ? Où veux-tu en venir ? Et surtout, comment peut-on faire autrement ?

Non, le positionnement absolu ce n'est pas "mal". Il sert parfois quand on a vraiment besoin de positionner au pixel près. Vous

pouvez l'utiliser dans certains de vos projets, mais autant que possible, préférez l'autre méthode : le positionnement relatif.

Le positionnement relatif, cela consiste à expliquer comment les widgets sont agencés les uns par rapport aux autres, plutôt que d'utiliser une position en pixels. Par exemple, on peut dire "Le bouton 1 est en-dessous du bouton 2, qui est à gauche du bouton 3".

Le positionnement relatif est géré par ce qu'on appelle les *layouts* avec Qt. Ce sont des conteneurs de widgets.

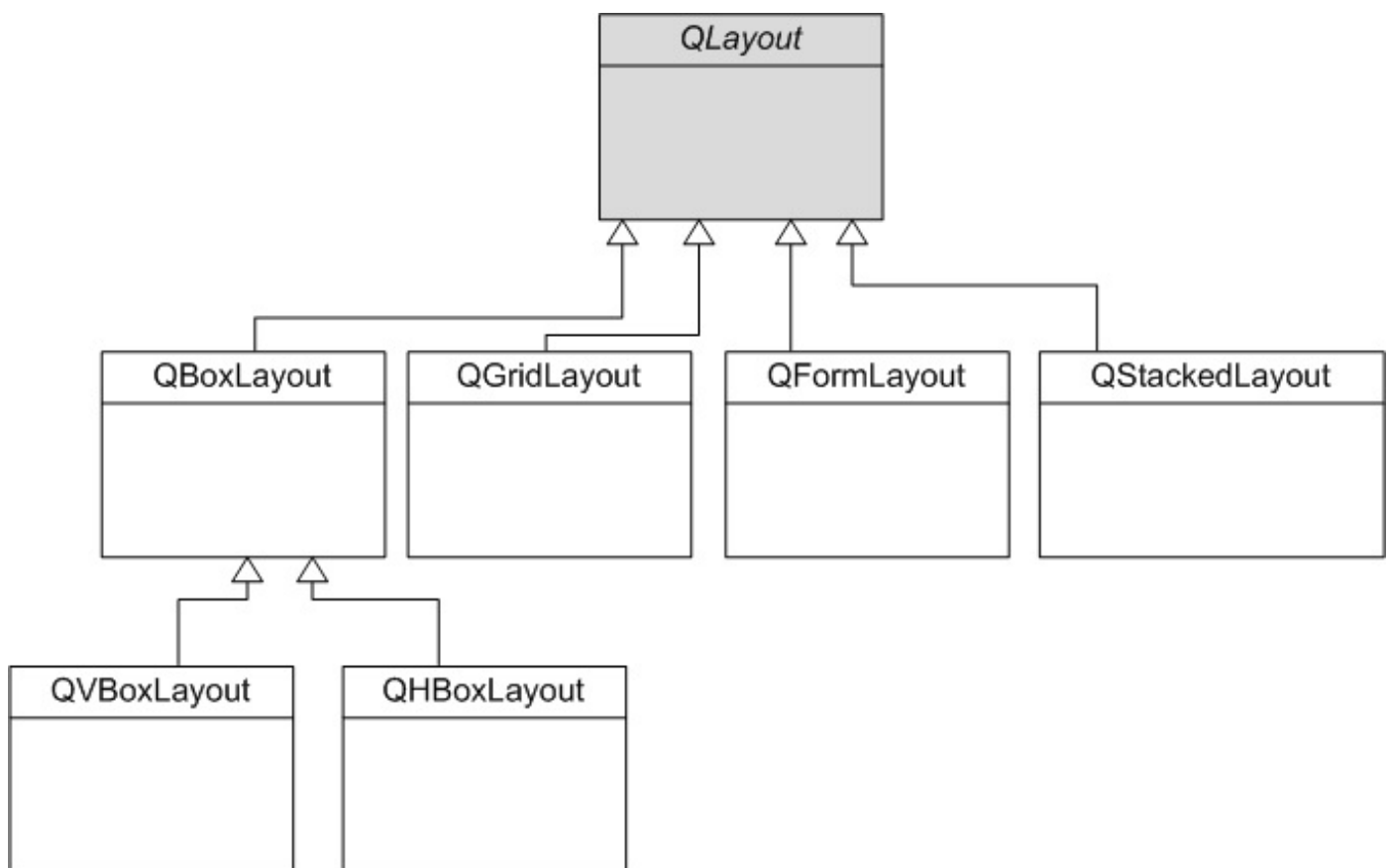
C'est justement l'objet principal de ce chapitre 😊

L'architecture des classes de layout

Pour positionner intelligemment nos widgets, nous allons utiliser des classes de Qt gérant les layouts.

Il existe par exemple des classes gérant le positionnement horizontal et vertical des widgets (ce que nous allons étudier en premier), ou encore le positionnement sous forme de grille.

Pour que vous y voyiez plus clair, je vous propose de regarder ce schéma de mon cru :



Ce sont les classes gérant les layouts de Qt.

Toutes les classes héritent de la classe de base QLayout.

On compte donc en gros les classes :

- QHBoxLayout
- QHBoxLayout
- QVBoxLayout
- QVBoxLayout
- QGridLayout
- QFormLayout
- QStackedLayout

Nous allons étudier chacune de ces classes dans ce chapitre, à l'exception de QStackedLayout (gestion des widgets sur plusieurs pages) qui est un peu trop complexe pour qu'on puisse travailler dessus ici. On utilisera plutôt des widgets qui le réutilisent, comme QWizard qui permet de créer des assistants.



Euh... Mais pourquoi tu as écrit *QLayout* en italique, et pourquoi tu as grisé la classe ? 🤔

QLayout est une classe abstraite. Souvenez-vous du chapitre sur le [polymorphisme](#), nous y avons vu qu'il est possible de créer des classes avec des [méthodes virtuelles pures](#). Ces classes sont dites *abstraites* parce qu'on ne peut pas instancier d'objet de ce type.

L'utilisation de classes abstraites par Qt pour les layouts est un exemple typique. Tous les layouts ont des propriétés communes et des méthodes qui effectuent la même action mais de manière différente. Afin de représenter toutes les actions possibles dans une seule *interface*, les développeurs ont choisi d'utiliser une classe abstraite. Voilà donc un exemple concret pour illustrer ce point de théorie un peu difficile. 😊

Les layouts horizontaux et verticaux

Attaquons sans plus tarder l'étude de nos premiers layouts (les plus simples), vous allez mieux comprendre à quoi tout cela sert 😊

Nous allons travailler sur 2 classes :

- [QHBoxLayout](#)
- [QVBoxLayout](#)

QHBoxLayout et QVBoxLayout héritent de [QBoxLayout](#). Ce sont des classes très similaires (la doc Qt parle de "*convenience classes*", des classes qui sont là pour vous aider à aller plus vite mais qui sont en fait quasiment identiques à QBoxLayout). Nous n'allons pas utiliser QBoxLayout, mais juste ses classes filles QHBoxLayout et QVBoxLayout (ça revient au même).

Le layout horizontal

L'utilisation d'un layout se fait en 3 temps :

1. On crée les widgets
2. On crée le layout et on place les widgets dedans
3. On dit à la fenêtre d'utiliser le layout qu'on a créé

1/ Créer les widgets

Pour les besoins de ce tutoriel, nous allons créer plusieurs boutons de type QPushButton :

Code : C++

```
QPushButton *bouton1 = new QPushButton("Bonjour");
QPushButton *bouton2 = new QPushButton("les");
QPushButton *bouton3 = new QPushButton("Zéros");
```

Vous remarquerez que j'utilise des pointeurs. En effet, j'aurais très bien pu faire sans pointeurs comme ceci :

Code : C++

```
QPushButton bouton1("Bonjour");
QPushButton bouton2("les");
QPushButton bouton3("Zéros");
```

... cette méthode a l'air plus simple, mais vous verrez que c'est plus pratique de travailler directement avec des pointeurs par la suite 😊

La différence entre ces 2 codes, c'est que bouton1 est un pointeur dans le premier code, tandis que c'est un objet dans le second code.

On va donc utiliser la première méthode avec les pointeurs.

Bon, on a 3 boutons, c'est bien. Mais les plus perspicaces d'entre vous auront remarqué qu'on n'a pas indiqué quelle était la fenêtre parente, comme on aurait fait avant :

Code : C++

```
QPushButton *bouton1 = new QPushButton("Bonjour", &fenetre);
```

On n'a pas fait comme ça, et c'est fait exprès justement. Nous n'allons pas placer les boutons dans la fenêtre directement, mais dans un conteneur : le layout.

2/ Créer le layout et placer les widgets dedans

Créons justement ce layout, un layout horizontal :

Code : C++

```
QHBoxLayout *layout = new QHBoxLayout;
```

Le constructeur de cette classe est simple, on n'a pas besoin d'indiquer de paramètre.

Maintenant que notre layout est créé, rajoutons nos widgets à l'intérieur :

Code : C++

```
layout->addWidget(bouton1);  
layout->addWidget(bouton2);  
layout->addWidget(bouton3);
```

La méthode addWidget du layout attend que vous lui donniez en paramètre un pointeur vers le widget à ajouter au conteneur. Voilà pourquoi je vous ai fait utiliser des pointeurs (sinon il aurait fallu écrire `layout->addWidget(&bouton1);` à chaque fois).

3/ Indiquer à la fenêtre d'utiliser le layout

Maintenant, dernière chose : il faut placer le layout dans la fenêtre. Il faut dire à la fenêtre : *"tu vas utiliser ce layout, qui contient mes widgets"*.

Code : C++

```
fenetre.setLayout(layout);
```


La méthode `setLayout` de la fenêtre attend un pointeur vers le layout à utiliser.
Et voilà, notre fenêtre contient maintenant notre layout, qui contient les widgets. Le layout se chargera d'organiser les widgets horizontalement tout seul.

Résumé du code

Voici le code complet de notre fichier `main.cpp` :

Code : C++

```
#include <QApplication>
#include <QPushButton>
#include <QHBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(bouton1);
    layout->addWidget(bouton2);
    layout->addWidget(bouton3);

    fenetre.setLayout(layout);

    fenetre.show();

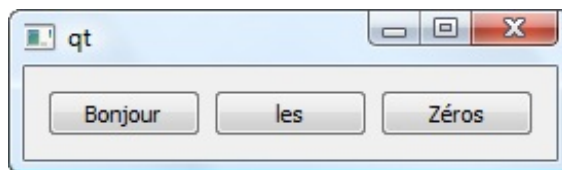
    return app.exec();
}
```

J'ai surligné les principales nouveautés.

En particulier, comme d'hab' lorsque vous utilisez une nouvelle classe Qt, pensez à l'inclure au début de votre code : `#include <QHBoxLayout>`

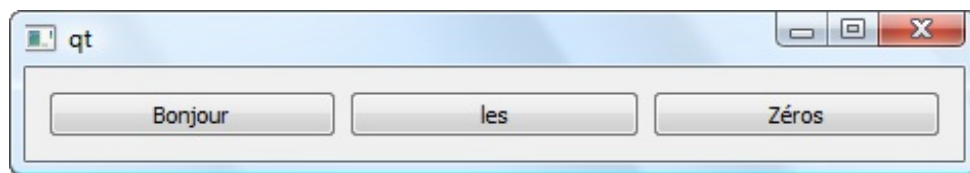
Résultat

Voilà à quoi ressemble la fenêtre maintenant que l'on utilise un layout horizontal :



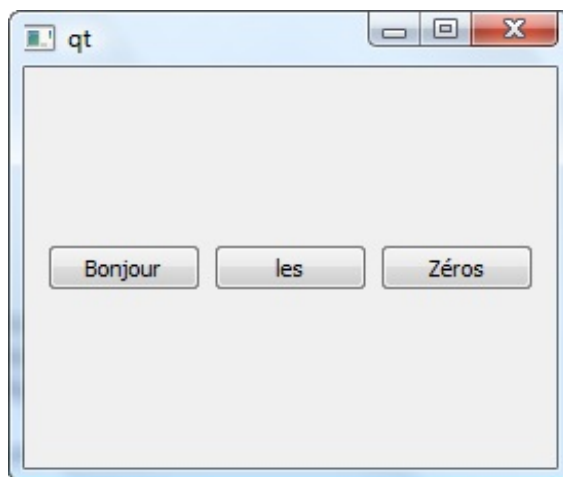
Les boutons sont automatiquement disposés de manière horizontale ! 😊

L'intérêt principal du layout, c'est son comportement face aux redimensionnements de la fenêtre.
Essayons de l'élargir :



Les boutons continuent de prendre l'espace en largeur.

On peut aussi l'agrandir en hauteur :

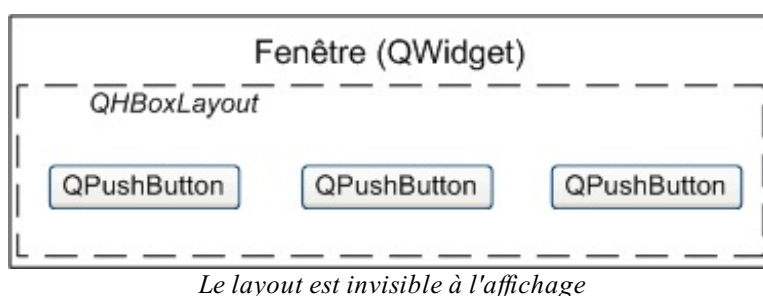


On remarque que les widgets restent centrés verticalement.

Vous pouvez aussi essayer de réduire la taille de la fenêtre. On vous interdira de la réduire si les boutons ne peuvent plus être affichés, ce qui vous garantit que les boutons ne risquent plus de disparaître comme avant ! 😊

Schéma des conteneurs

En résumé, la fenêtre contient le layout qui contient les widgets. Le layout se charge d'organiser les widgets. Schématiquement, ça se passe donc comme ça :



On vient de voir le layout QHBoxLayout qui organise les widgets horizontalement.

Il y en a un autre qui les organise verticalement (c'est quasiment la même chose) : QVBoxLayout.

Le layout vertical

Pour utiliser un layout vertical, il suffit de remplacer QHBoxLayout par QVBoxLayout dans le code précédent. Oui oui, c'est aussi simple que ça 😊

Code : C++

```
#include <QApplication>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(bouton1);
    layout->addWidget(bouton2);
    layout->addWidget(bouton3);

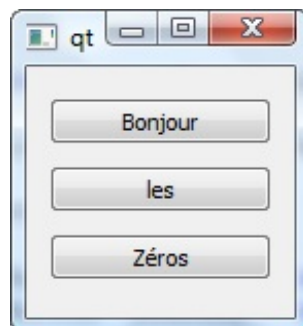
    fenetre.setLayout(layout);

    fenetre.show();

    return app.exec();
}
```

N'oubliez pas d'inclure QVBoxLayout.

Compilez et exécutez ce code, et admirez le résultat :



Amusez-vous à redimensionner la fenêtre. Vous voyez là encore que la layout adapte les widgets qu'il contient à toutes les dimensions. Il empêche en particulier la fenêtre de devenir trop petite, ce qui aurait empêché l'affichage des boutons.

La suppression automatique des widgets



Eh ! Je viens de me rendre compte que tu fais des new dans tes codes, mais il n'y a pas de delete ! Si tu alloues des objets sans les supprimer, ils vont pas rester en mémoire ?

Si, mais comme je vous l'avais dit plus tôt, Qt est intelligent 😊

En fait, les widgets sont placés dans un layout, qui est lui-même placé dans la fenêtre. Lorsque la fenêtre est supprimée (ici à la fin du programme), tous les widgets contenus dans son layout sont supprimés par Qt. C'est donc Qt qui se charge de faire les delete pour nous.

Bien, vous devriez commencer à comprendre comment fonctionnent les layouts 😊

Comme on l'a vu au début du chapitre, il y a de nombreux layouts, qui ont chacun leurs spécificités ! Intéressons-nous maintenant au puissant (mais complexe) `QGridLayout`.

Le layout de grille

Les layouts horizontaux et verticaux sont gentils, mais il ne permettent pas de créer des dispositions très complexes sur votre fenêtre.

C'est là qu'entre en jeu `QGridLayout`, qui est en fait un peu un assemblage de `QHBoxLayout` et `QVBoxLayout`. Il s'agit d'une disposition en grille, comme un tableau avec des lignes et des colonnes.

Schéma de la grille

Il faut imaginer que votre fenêtre peut être découpée sous la forme d'une grille avec une infinité de cases, comme ceci :

0, 0	0, 1	0, 2	...
1, 0	1, 1	1, 2	...
2, 0	2, 1	2, 2	...
...

Si on veut placer un widget en haut à gauche, il faudra le placer à la case de coordonnées (0, 0).

Si on veut en placer un autre en-dessous, il faudra utiliser les coordonnées (1, 0).

Ainsi de suite 😊

Utilisation basique de la grille

Essayons d'utiliser un `QGridLayout` simplement pour commencer (oui parce qu'on peut aussi l'utiliser de manière compliquée 😊).

Nous allons placer un bouton en haut à gauche, un à sa droite et un en-dessous.

La seule différence réside en fait dans l'appel à la méthode `addWidget`. Celle-ci accepte 2 paramètres supplémentaires : les coordonnées où placer le widget sur la grille.

Code : C++

```
#include <QApplication>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
```

```

QPushButton *bouton1 = new QPushButton("Bonjour");
QPushButton *bouton2 = new QPushButton("les");
QPushButton *bouton3 = new QPushButton("Zéros");

QGridLayout *layout = new QGridLayout;
layout->addWidget(bouton1, 0, 0);
layout->addWidget(bouton2, 0, 1);
layout->addWidget(bouton3, 1, 0);

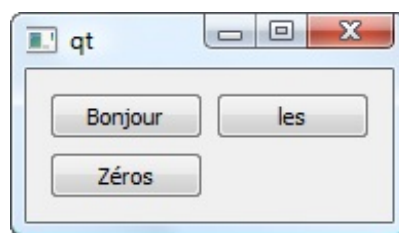
fenetre.setLayout(layout);

fenetre.show();

return app.exec();
}

```

Résultat :



Si vous comparez avec le schéma de la grille que j'ai fait plus haut, vous voyez que les boutons ont bien été disposés selon les bonnes coordonnées.



D'ailleurs en parlant du schéma plus haut, il y a un truc que je comprends pas, c'est tous ces points de suspension "...". Ça veut dire que la taille de la grille est infinie ? Dans ce cas, comment je fais pour placer un bouton en bas à droite ?

Qt "sait" quel est le widget à mettre en bas à droite en fonction des coordonnées des autres widgets. Le widget qui a les coordonnées les plus élevées sera placé en bas à droite.

Petit test, rajoutons un bouton aux coordonnées (1, 1) :

Code : C++

```

#include <QApplication>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

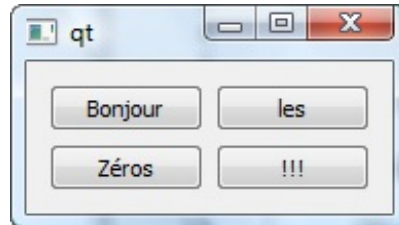
    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");
    QPushButton *bouton4 = new QPushButton("!!!");

    QGridLayout *layout = new QGridLayout;
    layout->addWidget(bouton1, 0, 0);
    layout->addWidget(bouton2, 0, 1);
    layout->addWidget(bouton3, 1, 0);
    layout->addWidget(bouton4, 1, 1);
}

```

```
fenetre.setLayout(layout);  
fenetre.show();  
return app.exec();  
}
```

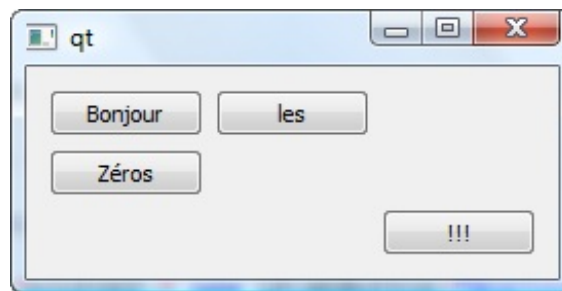
Résultat :



Si on veut, on peut aussi décaler le bouton encore plus en bas à droite dans une nouvelle ligne et une nouvelle colonne :

Code : C++

```
layout->addWidget(bouton4, 2, 2);
```



C'est compris ? 😊

Un widget qui occupe plusieurs cases

L'avantage de la disposition en grille, c'est qu'on peut faire en sorte qu'un widget occupe plusieurs cases à la fois. On parle de **spanning** (ceux qui font du HTML doivent avoir entendu parler des attributs *rowspan* et *colspan* sur les tableaux).

Pour faire cela, il faut appeler une version surchargée de `addWidget` qui accepte 2 paramètres supplémentaires : le `rowSpan` et le `columnSpan`.

- `rowSpan` : nombre de lignes qu'occupe le widget (par défaut 1)
- `columnSpan` : nombre de colonnes qu'occupe le widget (par défaut 1)

Imaginons un widget placé en haut à gauche, aux coordonnées (0, 0). Si on lui donne un `rowSpan` de 2, il occupera alors l'espace suivant :

rowSpan = 2			...
			...
			...
...

Si on lui donne un `columnSpan` de 3, il occupera cet espace :

columnSpan = 3			...
			...
			...
...



L'espace pris par le widget au final dépend de la nature du widget (les boutons s'agrandissent en largeur mais pas en hauteur par exemple), et dépend du nombre de widgets sur la grille. En pratiquant vous allez rapidement comprendre comment ça fonctionne.

Essayons de faire en sorte que le bouton "Zéros" prenne 2 colonnes de largeur :

Code : C++

```
#include <QApplication>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

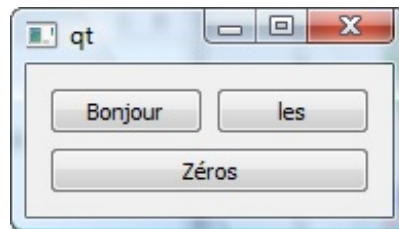
    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");
```

```
QGridLayout *layout = new QGridLayout;  
layout->addWidget(bouton1, 0, 0);  
layout->addWidget(bouton2, 0, 1);  
layout->addWidget(bouton3, 1, 0, 1, 2);  
  
fenetre.setLayout(layout);  
  
fenetre.show();  
  
return app.exec();  
}
```

Notez la ligne : `layout->addWidget(bouton3, 1, 0, 1, 2);`

Les 2 derniers paramètres correspondent respectivement au `rowSpan` et au `columnSpan`. Le `rowSpan` est ici de 1, c'est la valeur par défaut on ne change donc rien, mais le `columnSpan` est de 2.

Le bouton va donc "occuper" 2 colonnes :



Essayez en revanche de monter le `columnSpan` à 3 : vous ne verrez aucun changement.

En effet, il aurait fallu qu'il y ait un troisième widget sur la première ligne pour que le `columnSpan` puisse fonctionner.

Faites des tests avec le spanning pour vous assurer que vous avez bien compris comment ça marche 😊

Le layout de formulaire

Le layout de formulaire `QFormLayout` est un layout assez récent spécialement fait pour les fenêtres qui contiennent des formulaires.

Un formulaire est en général une suite de libellés ("Votre prénom :") associés à des champs de formulaire (zone de texte par exemple) :

Votre prénom :	<input type="text" value="Anna"/>
Votre nom :	<input type="text" value="Conda"/>
Votre âge :	<input type="text" value="26"/>

Normalement, pour écrire du texte dans la fenêtre, on utilise le widget `QLabel` (libellé), dont on parlera plus en détail dans le prochain chapitre.

L'avantage du layout que nous allons utiliser, c'est qu'il simplifie notre travail en créant automatiquement des `QLabel` pour nous.



Vous noterez d'ailleurs que la disposition correspond à celle d'un `QGridLayout` à 2 colonnes et plusieurs lignes. En effet, le `QFormLayout` n'est en fait rien d'autre qu'une version spéciale du `QGridLayout` pour les formulaires, avec quelques particularités : il s'adapte en fonction des habitudes des OS, pour certains les libellés sont alignés à gauche, pour d'autres ils sont alignés à droite, etc.

L'utilisation d'un `QFormLayout` est très simple. La différence, c'est qu'au lieu d'utiliser une méthode `addWidget`, nous allons

utiliser une méthode `addRow` qui prend 2 paramètres :

- Le texte du libellé
- Un pointeur vers le champ du formulaire

Pour faire simple, nous allons créer 3 champs de formulaire de type "Zone de texte à une ligne" (`QLineEdit`), puis nous allons les placer dans un `QFormLayout` au moyen de la méthode `addRow` :

Code : C++

```
#include <QApplication>
#include <QLineEdit>
#include <QFormLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;

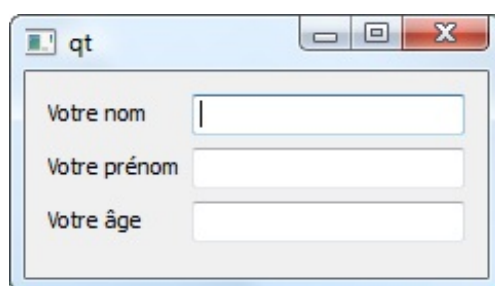
    QFormLayout *layout = new QFormLayout;
    layout->addRow("Votre nom", nom);
    layout->addRow("Votre prénom", prenom);
    layout->addRow("Votre âge", age);

    fenetre.setLayout(layout);

    fenetre.show();

    return app.exec();
}
```

Résultat :



Sympa, non ? 😊

On peut aussi définir des raccourcis clavier pour accéder rapidement aux champs du formulaire. Pour ce faire, placez un symbole "&" devant la lettre du libellé que vous voulez transformer en raccourci.

Explication en image (euh, en code) :

Code : C++

```
layout->addRow("Votre &nom", nom);
layout->addRow("Votre &prénom", prenom);
```

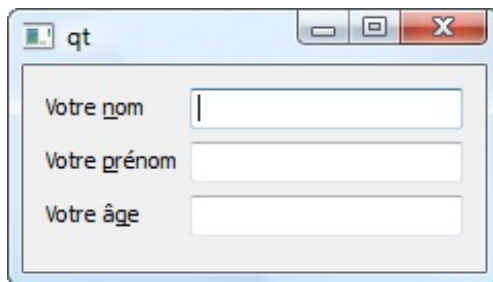
```
layout->addRow("Vot&re â&ge", age);
```

La lettre "p" est désormais un raccourci vers le champ du prénom.

"n" pour le champ nom.

"g" pour le champ âge.

L'utilisation du raccourci dépend de votre système d'exploitation. Sous Windows, il faut faire Alt puis la touche raccourci. Lorsque vous appuyez sur Alt, les lettres raccourcis apparaissent soulignées :



Faites Alt + N pour accéder directement au champ du nom ! 😊



Souvenez-vous de ce symbole &, il est très souvent utilisé en GUI Design (design de fenêtre) pour indiquer quelle lettre sert de raccourci. On le réutilisera notamment pour avoir des raccourcis dans les menus de la fenêtre.

Ah, et si vous voulez par contre vraiment afficher un symbole & dans un libellé, tapez-en deux : "&&".
Exemple : "Bonnie && Clyde".

Combiner les layouts

Avant de terminer ce chapitre, il me semble important que nous jetions un oeil aux **layouts combinés**, une fonctionnalité qui va vous faire comprendre toute la puissance des layouts.

Commençons comme il se doit par une question que vous devriez vous poser :



Les layouts c'est bien joli, mais c'est pas un peu limité ? Si je veux faire une fenêtre un peu complexe, ce n'est pas à grands coups de QVBoxLayout ou même de QGridLayout que je vais m'en sortir !

C'est vrai que mettre ses widgets les uns en-dessous des autres peut sembler limité. Même la grille fait un peu "rigide", je reconnais.

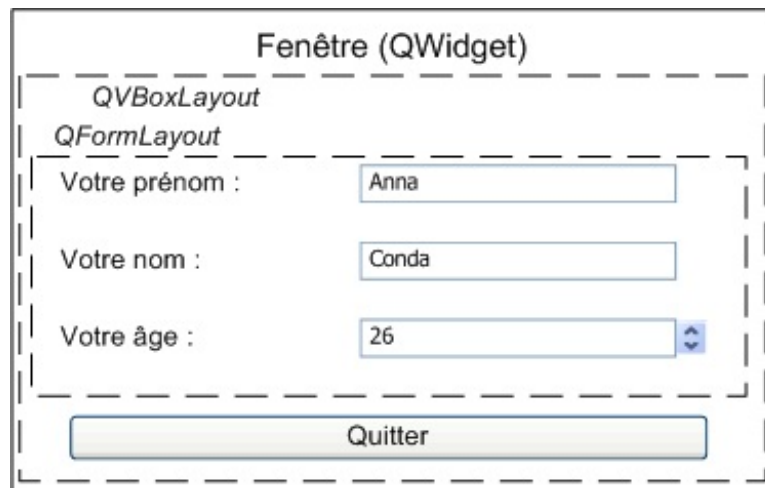
Mais rassurez-vous, tout a été pensé. La magie apparaît lorsque nous commençons à combiner les layouts, c'est-à-dire à placer un layout dans un autre layout.

Un cas concret

Prenons par exemple notre joli formulaire. Supposons que l'on veuille ajouter un bouton "Quitter". Si vous voulez placer ce bouton en bas du formulaire, comment faire ?

Il va falloir d'abord créer un layout vertical (QVBoxLayout), et placer à l'intérieur notre layout de formulaire *puis* notre bouton "Quitter".

Cela donne le schéma suivant :



On voit que notre QVBoxLayout contient 2 choses, dans l'ordre :

1. Un QFormLayout (qui contient lui-même d'autres widgets)
2. Un QPushButton

Un layout peut donc contenir aussi bien des layouts que des widgets.

Utilisation de addLayout

Pour insérer un layout dans un autre, on utilise addLayout au lieu de addWidget (c'est logique me direz-vous 🤔).

Voici un bon petit code pour se faire la main :

Code : C++

```
#include <QApplication>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QFormLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    // Création du layout de formulaire et de ses widgets

    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;

    QFormLayout *formLayout = new QFormLayout;
    formLayout->addRow("Votre &nom", nom);
    formLayout->addRow("Votre &prénom", prenom);
    formLayout->addRow("Votre &âge", age);

    // Création du layout principal de la fenêtre (vertical)

    QVBoxLayout *layoutPrincipale = new QVBoxLayout;
    layoutPrincipale->addLayout(formLayout); // Ajout du layout de
```

formulaire

```
QPushButton *boutonQuitter = new QPushButton("Quitter");
QWidget::connect(boutonQuitter, SIGNAL(clicked()), &app,
SLOT(quit()));
layoutPrincipal->addWidget(boutonQuitter); // Ajout du bouton

fenetre.setLayout(layoutPrincipal);

fenetre.show();

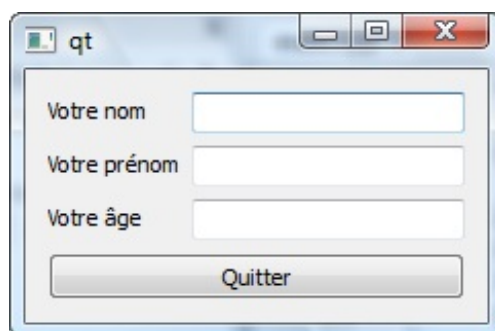
return app.exec();
}
```

J'ai surligné les ajouts au layout vertical principal :

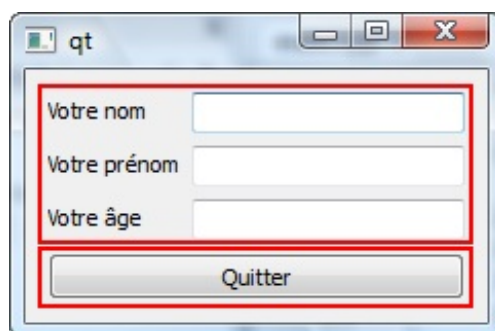
- L'ajout du sous-layout de formulaire (addLayout)
- L'ajout du bouton (addWidget)

Vous remarquerez que je fais les choses un peu dans l'ordre inverse : d'abord je crée les widgets et layouts "enfants" (le QFormLayout), et ensuite je crée le layout principal (le QVBoxLayout) et j'y ajoute le layout enfant que j'ai créé.

Au final, la fenêtre qui apparaît est la suivante :

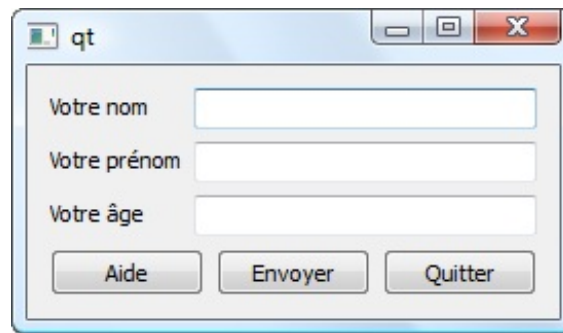


On ne le voit pas, mais la fenêtre contient d'abord un QVBoxLayout, qui contient lui-même un layout de formulaire et un bouton :



Exercice

Essayez d'obtenir le rendu suivant :



Si vous voulez mettre plusieurs boutons en bas sur la même ligne, vous pouvez créer un QHBoxLayout et ajouter ce QHBoxLayout au QVBoxLayout !

Vous pouvez aussi utiliser plus simplement un QGridLayout en utilisant un columnSpan. En effet, un QGridLayout n'est rien d'autre qu'un assemblage de QVBoxLayout et de QHBoxLayout.

Plusieurs méthodes sont donc possibles, libre à vous d'utiliser un QGridLayout ou des QVBoxLayout et QHBoxLayout.

Ce ne devrait pas être un exercice difficile si vous avez bien suivi ce chapitre. Ce sera en tout cas l'occasion de vous assurer que vous avez bien compris 😊



Dans mon exemple, les boutons "Aide" et "Envoyer" ne font rien (je n'ai pas géré de signaux et de slots pour eux). Le résultat que vous devez obtenir est juste visuel, n'essayez pas de tenter d'envoyer le formulaire sur internet et de le stocker dans une base de données, il est un peu trop tôt encore 🤪

Les layouts sont la base du positionnement de widgets en GUI Design. Ils nous donnent un maximum de flexibilité pour que nos fenêtres s'adaptent à toutes les conditions.

Bien entendu, je vous mentirais si je vous disais qu'absolument tout le monde les utilise. Pour certains logiciels simples, il n'est parfois pas nécessaire de recourir aux layouts. Il est néanmoins recommandé de s'en servir autant que possible.

Nous n'avons pas pu absolument *tout* voir à propos des layouts. La différence, c'est que maintenant je vous ai appris à vous servir de la doc et vous pouvez aller compléter ce que vous savez si besoin est.



Je vous recommande de lire leur page d'explication générale sur les layouts puis de regarder les différentes classes de layouts. N'oubliez pas de consulter les classes parentes à chaque fois, ce sont souvent elles qui contiennent les méthodes et attributs qui semblent manquer.

Jetez un oeil aux "stretch factors", qui permettent de définir des tailles proportionnelles pour les widgets, ainsi qu'à l'alignement des widgets.

Dans le prochain chapitre, nous passerons en revue la plupart des widgets courants et simples. En effet, cela fait un moment que je vous fais utiliser pour le besoin du cours quelques widgets comme les boutons et les champs de texte, mais il est maintenant temps de faire un tour d'horizon plus général pour que vous sachiez quels sont les principaux widgets qui peuvent peupler une fenêtre.

Les principaux widgets

Voilà un moment que nous avons commencé à nous intéresser à Qt, je vous parle en long en large et en travers de widgets, mais jusqu'ici nous n'avions toujours pas pris le temps de faire un tour d'horizon rapide de ce qui existait.

C'était voulu. Je voulais dans un premier temps vous faire manipuler un ou deux widgets simples pour vous faire comprendre les concepts de base comme :

- La création de la fenêtre
- Les signaux et les slots
- Les layouts

Il est maintenant temps de faire une "pause" et de regarder ce qui existe comme widgets. Nous étudierons cependant seulement les principaux widgets ici.

Pourquoi ne les verrons-nous pas tous ? Parce qu'il existe un grand nombre de widgets et que certains sont rarement utilisés. D'autres sont parfois tellement complexes qu'ils nécessiteront un chapitre entier pour les étudier.

Néanmoins, avec ce que vous allez voir, vous aurez largement de quoi faire pour être capables de créer la quasi-totalité des fenêtres que vous voulez ! 😊

Pour information, je me base sur la page "[liste des widgets](#)" (ici avec l'apparence de vista, mais peu importe l'apparence, ça sera adapté à votre OS).



Je ne compte pas remplacer la doc. Je vous inviterai donc à consulter la doc à chaque fois pour en savoir plus. Mon rôle sera surtout de vous introduire à utiliser de manière basique ces widgets. Je vous fais confiance, je sais que vous saurez en faire une utilisation plus avancée si besoin est. 😊

Les fenêtres

Avec Qt, tout élément de la fenêtre est appelé un **widget**. La fenêtre elle-même est considérée comme un widget.

Dans le code, les widgets sont des classes qui héritent toujours de **QWidget** (directement ou indirectement). C'est donc une classe de base très importante, et vous aurez probablement très souvent besoin de lire la doc de cette classe.

Quelques rappels sur l'ouverture d'une fenêtre

Cela fait plusieurs chapitres que l'on crée une fenêtre dans nos programmes à l'aide d'un objet de type QWidget. Cela signifie-t-il que QWidget = Fenêtre ?

Non. En fait, un widget qui n'est contenu dans aucun autre widget est considéré comme une fenêtre.

Donc quand on fait juste ce code très simple :

Code : C++

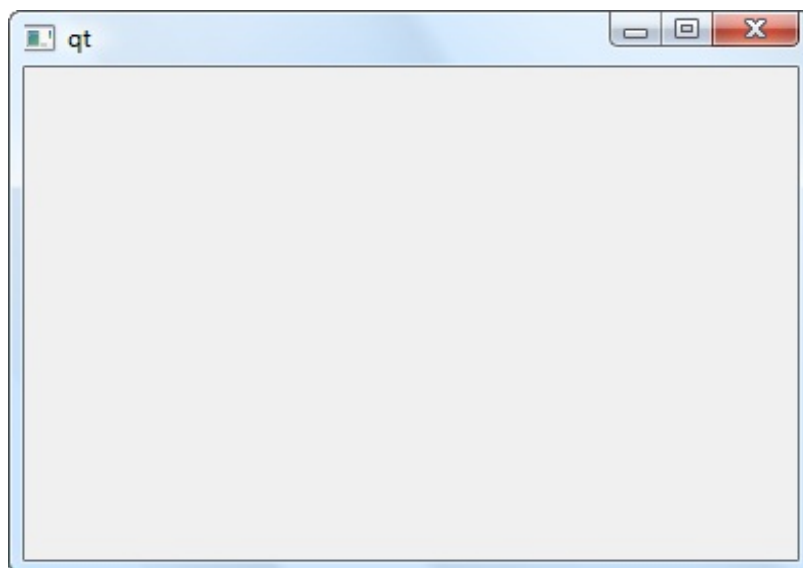
```
#include <QApplication>
#include <QWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    fenetre.show();

    return app.exec();
}
```

... cela affiche une fenêtre (vide) :



C'est comme cela que Qt fonctionne. C'est un peu déroutant au début, mais après on apprécie au contraire que ça ait été pensé comme ça.



Donc si je comprends bien, il n'y a pas de classe `QFenetre` ou quelque chose du genre ?

Tout à fait, il n'y a pas de classe du genre "`QFenetre`" car n'importe quel widget peut servir de fenêtre. Si vous vous souvenez bien, on avait créé un bouton dans les premiers exemples du cours sur Qt. On avait demandé à afficher ce bouton. Comme le bouton n'avait pas de parent, une fenêtre avait été ouverte :

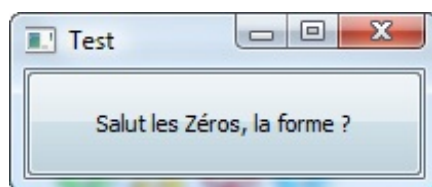
Code : C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.show();

    return app.exec();
}
```



Dans la pratique, on ne crée pas de fenêtre-bouton comme là. On crée d'abord une fenêtre, et on place ensuite des widgets à l'intérieur (ces widgets étant parfois organisés grâce aux layouts comme on l'a vu).

Code : C++

```
#include <QApplication>
#include <QWidget>
```

```
#include <QPushButton>

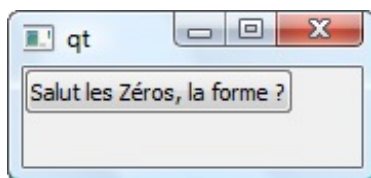
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    QPushButton bouton("Salut les Zéros, la forme ?", &fenetre);
    fenetre.show();

    return app.exec();
}
```



Note : je n'ai pas utilisé de layouts dans ce code pour le rendre court et simple. Le bouton est donc positionné de manière absolue au coin en haut à gauche, de coordonnées (0, 0).



Le QPushButton a donc pour parent un QWidget, car on a indiqué en second paramètre de son constructeur un pointeur vers le QWidget : `&fenetre`.

Le QWidget n'a pas de parent car on n'a pas envoyé de pointeur vers un autre widget dans son constructeur, donc c'est une fenêtre.

Ne confondez pas :



- **En termes C++** : une classe parente est une classe mère (quand on fait un héritage).
- **En termes Qt** : un widget parent est un widget qui en contient d'autres. Un widget fils est un widget qui n'en contient aucun autre.

Ici, je suis en train de parler de widgets parents en termes Qt.

Quelques classes particulières pour les fenêtres

Résumons ce que je viens de dire : tout widget peut servir de fenêtre.

C'est le widget qui n'a pas de parent qui sera considéré comme étant la fenêtre.

A ce titre, un QPushButton ou un QLineEdit peuvent être considérés comme des fenêtres s'ils n'ont pas de widget parent.

Toutefois, il y a 2 classes de widgets que j'aimerais mettre en valeur :

- **QMainWindow** : c'est un widget spécial qui permet de créer la fenêtre principale de l'application. Une fenêtre principale peut contenir des menus, une barre d'outils, une barre d'état, etc.
- **QDialog** : c'est une classe de base utilisée par toutes les classes de boîtes de dialogue qu'on a vues il y a quelques chapitres. On peut aussi s'en servir directement pour ouvrir des boîtes de dialogue personnalisées.

La fenêtre principale QMainWindow mérite un chapitre entier à elle toute seule. Et elle en aura un. Nous pourrons alors tranquillement passer en revue la gestion des menus, de la barre d'outils et de la barre d'état.

La fenêtre QDialog peut être utilisée pour ouvrir une boîte de dialogue personnalisée générique. Une boîte de dialogue est une fenêtre généralement de petite taille dans laquelle il y a peu d'informations.

La classe QDialog hérite de QWidget comme tout widget qui se respecte, et elle y est même très similaire. Elle y ajoute peu de choses, parmi lesquelles la gestion des *fenêtres modales* (une fenêtre par-dessus toutes les autres qui doit être remplie avant de pouvoir accéder aux autres fenêtres de l'application).

Nous allons ici étudier ce que l'on peut faire d'intéressant avec la classe de base QWidget qui permet déjà de réaliser la plupart des fenêtres que l'on veut.

Nous verrons ensuite ce qu'on peut faire avec les fenêtres de type QDialog. Quant à QMainWindow, ce sera pour un autre chapitre comme je vous l'ai dit. 😊

Une fenêtre avec QWidget

Pour commencer, je vous invite à ouvrir la [doc de QWidget](#) en même temps que vous lisez ce chapitre.

Vous remarquerez que QWidget est la classe mère d'un grrrrand nombre d'autres classes. 😊

Les QWidget disposent de beaucoup de propriétés et de méthodes. Donc tous les widgets disposent de ces propriétés et méthodes.

On peut découper les propriétés en 2 catégories :

- Celles qui valent pour tous les types de widgets et pour les fenêtres
- Celles qui n'ont de sens que pour les fenêtres

Jetons un oeil à celles qui me semblent les plus intéressantes. Pour avoir la liste complète, il faudra recourir à la doc, je ne compte pas tout répéter ici ! 😊

Les propriétés utilisables pour tous les types de widgets, y compris les fenêtres

Je vous fais une liste rapide pour extraire quelques propriétés qui pourraient vous intéresser. Pour savoir comment vous servir de toutes ces propriétés, lisez le prototype que vous donne la doc.



N'oubliez pas qu'on peut modifier une propriété en appelant une méthode du même nom commençant par "set". Par exemple, si la propriété est cursor, la méthode sera setCursor().

- **cursor** : curseur de la souris à afficher lors du survol du widget. La méthode setCursor attend que vous lui envoyiez un objet de type QCursor. Certains curseurs classiques (comme le sablier) sont prédéfinis dans une énumération. La doc vous fait un lien vers cette énumération.
- **enabled** : indique si le widget est activé, si on peut le modifier. Un widget désactivé est généralement grisé. Si vous appliquez setEnabled(false) à toute la fenêtre, c'est toute la fenêtre qui deviendra inutilisable.
- **height** : hauteur du widget.
- **size** : dimensions du widget. Vous devrez indiquer la largeur et la hauteur.
- **visible** : contrôle la visibilité du widget.
- **width** : largeur.

N'oubliez pas : pour modifier une de ces propriétés, préfixez la méthode par un "set". Exemple :

Code : C++

```
maFenetre.setWidth(200);
```

Ces propriétés sont donc valables pour tous les widgets, y compris les fenêtres. Si vous appliquez un `setWidth` sur un bouton, ça modifiera la largeur du bouton. Si vous appliquez cela sur une fenêtre, c'est la largeur de la fenêtre qui sera modifiée.

Les propriétés utilisables uniquement sur les fenêtres

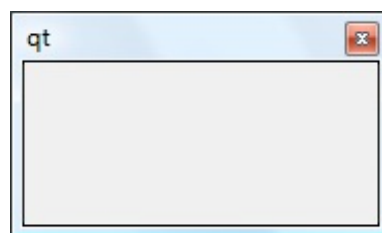
Ces propriétés sont faciles à reconnaître, elles commencent toutes par "window". Elles n'ont de sens que si elles sont appliquées aux fenêtres.

- **windowFlags** : une série d'options contrôlant le comportement de la fenêtre. Il faut consulter l'énumération `Qt::WindowType` pour savoir les différents types disponibles. Vous pouvez aussi consulter l'exemple Window Flags du programme "Qt Examples and Demos".

Par exemple pour afficher une fenêtre de type "Outil" avec une petite croix et pas de possibilité d'agrandissement ou de réduction :

Code : C++

```
fenetre.setWindowFlags (Qt::Tool);
```



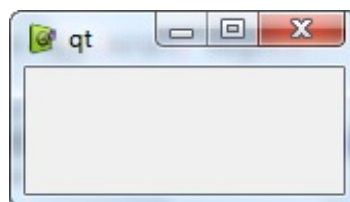
Une fenêtre de type "Tool"

C'est par là aussi qu'on passe pour que la fenêtre reste par-dessus toutes les autres fenêtres du système (avec le flag `Qt::WindowStaysOnTopHint`).

- **windowIcon** : l'icône de la fenêtre. Il faut envoyer un objet de type `QIcon`, qui lui-même accepte un nom de fichier à charger. Cela donne le code suivant pour charger le fichier `icone.png` situé dans le même dossier que l'application :

Code : C++

```
fenetre.setWindowIcon (QIcon ("icone.png")) ;
```

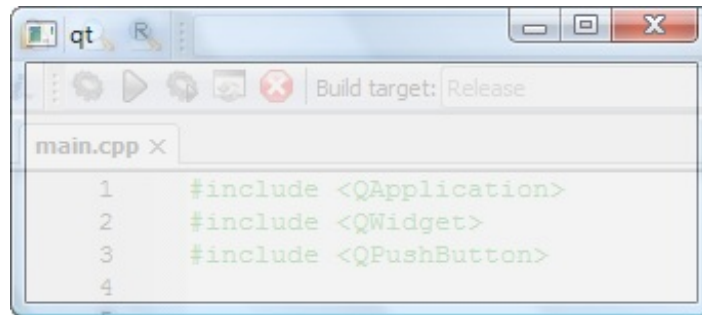


Une icône pour la fenêtre

- **windowOpacity** : contrôle la transparence de la fenêtre (ne fonctionne pas sur tous les OS). La valeur à envoyer est un nombre décimal compris entre 0 (transparent) et 1 (complètement opaque). Ici, avec la valeur 0.8 :

Code : C++

```
fenetre.setWindowOpacity(0.8);
```

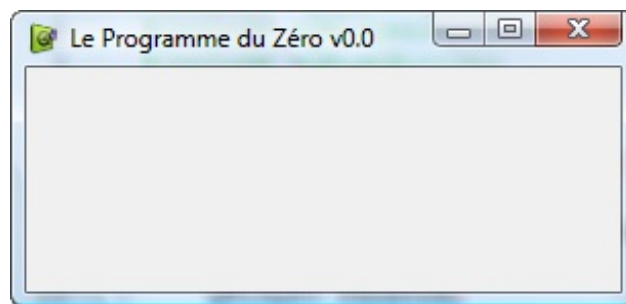


Une fenêtre transparente

- **windowTitle** : le titre de la fenêtre, affiché en haut.

Code : C++

```
fenetre.setWindowTitle("Le Programme du Zéro v0.0");
```



Une fenêtre avec un titre

Une fenêtre avec QDialog

QDialog est un widget spécialement créé pour générer des fenêtres de type "boîte de dialogue".



Quelle est la différence avec une fenêtre créée à partir d'un QWidget ? 🤔

En général les QDialog sont des petites fenêtres secondaires : des boîtes de dialogue. Elles proposent le plus souvent un choix simple entre :

- Valider
- Annuler

Les QDialog sont rarement utilisées pour gérer la fenêtre principale. Pour la fenêtre principale on préfère utiliser QWidget, ou carrément QMainWindow si on a besoin de l'artillerie lourde.

Les QDialog peuvent être de 2 types :

- **Modales** : on ne peut pas accéder aux autres fenêtres de l'application lorsqu'elles sont ouvertes.
- **Non modales** : on peut toujours accéder aux autres fenêtres.

Par défaut, les QDialog sont modales.

Elles disposent en effet d'une méthode exec() qui ouvre la boîte de dialogue de manière modale. Il s'avère d'ailleurs qu'exec() est un slot (très pratique pour effectuer une connexion ça !).

Je vous propose d'essayer de pratiquer de la manière suivante : nous allons ouvrir une fenêtre principale QWidget qui contiendra un bouton. Lorsqu'on cliquera sur ce bouton, il ouvrira une fenêtre secondaire de type QDialog.

Notre objectif est d'ouvrir une fenêtre secondaire après un clic sur un bouton de la fenêtre principale. La fenêtre secondaire, de type QDialog, affichera juste une image pour cet exemple.

Code : C++

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    QPushButton *bouton = new QPushButton("Ouvrir la fenêtre",
&fenetre);

    QDialog secondeFenetre (&fenetre);
    QVBoxLayout *layout = new QVBoxLayout;
    QLabel *image = new QLabel(&secondeFenetre);
    image->setPixmap(QPixmap("icone.png"));
    layout->addWidget(image);
    secondeFenetre.setLayout(layout);

    QWidget::connect(bouton, SIGNAL(clicked()), &secondeFenetre,
SLOT(exec()));
    fenetre.show();

    return app.exec();
}
```

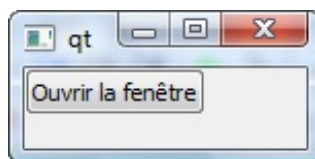
Mon code est indenté de manière bizarroïde je sais. Je trouve que c'est plus lisible : vous pouvez mieux voir comme cela à quelles fenêtres se rapportent les opérations que je fais.



Vous voyez ainsi immédiatement que dans la première fenêtre je n'ai fait que placer un bouton, tandis que dans la seconde j'ai mis un QLabel affichant une image que j'ai placée dans un QVBoxLayout.

D'autre part, j'ai tout fait dans le main pour cet exemple, mais dans la pratique, comme nous le verrons dans les TP, on a en général un fichier .cpp par fenêtre, c'est plus facile à gérer.

Au départ, la fenêtre principale s'affiche, comme ceci :



Si vous cliquez sur le bouton, la boîte de dialogue s'ouvre :



Comme elle est modale, vous remarquerez que vous ne pouvez pas accéder à la fenêtre principale tant qu'elle est ouverte.

Bon intéressons-nous au code. J'ai surligné les 2 lignes qui me paraissent les plus pertinentes :

- **Ligne 12** : la création de la QDialog. Rien de bien extraordinaire à première vue, mais si vous regardez bien vous devriez voir que j'ai mentionné dans le constructeur l'adresse de la fenêtre parente. Cela permet à la QDialog de savoir quelle est la fenêtre qui l'a appelée. Entre autres choses, la QDialog se placera automatiquement de manière centrée par rapport à sa fenêtre mère.
- **Ligne 20** : je connecte le clic sur le bouton à la méthode exec() de la QDialog pour ouvrir la boîte de dialogue (de manière modale).

Cela devrait vous donner des bases suffisantes pour désormais savoir comment ouvrir des fenêtres secondaires de type QDialog. Nous n'avons cependant pas tout vu sur cette classe : on peut rendre les QDialog non modales ou encore utiliser les slots accept() et reject() pour les connecter respectivement à des boutons "OK" et "Annuler" et ainsi informer la fenêtre parente afin qu'elle sache si l'opération a été validée ou refusée par l'utilisateur.

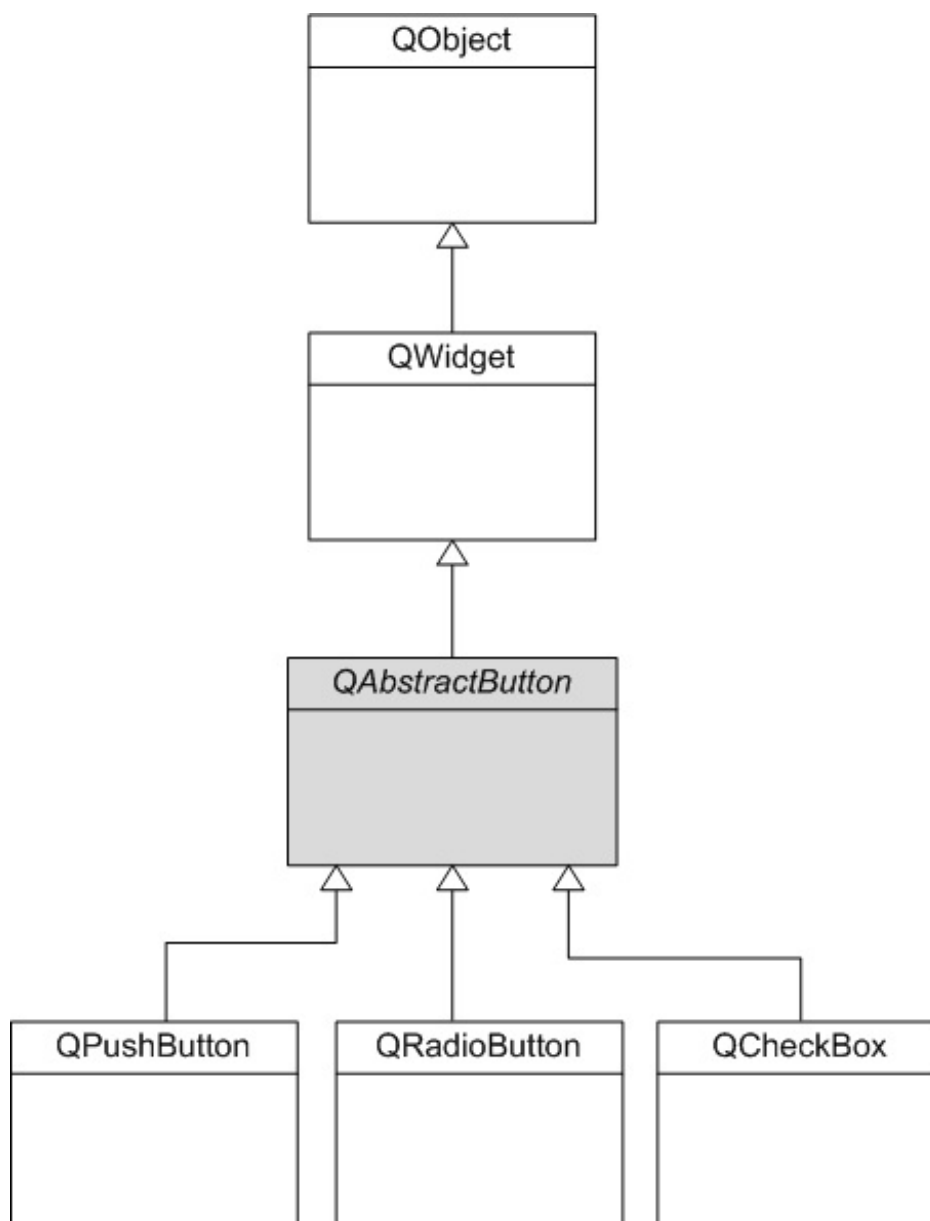
Pour savoir faire tout cela, vous savez ce qu'il vous reste à faire. Tout est dans la doc. 😊

Les boutons

Nous allons maintenant étudier la catégorie des widgets "boutons". Nous allons passer en revue :

- **QPushButton** : un bouton classique, que vous avez déjà largement eu l'occasion de manipuler.
- **QRadioButton** : un bouton "radio", pour un choix à faire parmi une liste.
- **QCheckBox** : une case à cocher (on considère que c'est un bouton en GUI Design).

Tous ces widgets héritent de [QAbstractButton](#) qui lui-même hérite de QWidget, qui finalement hérite de QObject :

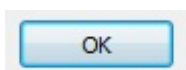


Comme l'indique son nom, `QAbstractButton` est une classe abstraite. Si vous vous souvenez des épisodes précédents de notre passionnant feuilleton, une classe abstraite est une classe... qu'on ne peut pas instancier, bravo ! 😊

On ne peut donc pas créer d'objets de type `QAbstractButton`, il faut forcément utiliser une des classes filles. `QAbstractButton` sert donc juste de modèle de base pour ses classes filles.

QPushButton : un bouton

Le `QPushButton` est l'élément le plus classique et le plus commun des fenêtres :



Je ne vous fais pas l'offense de vous expliquer à quoi sert un bouton 🤪

Commençons par un rappel important, indiqué par la doc : `QPushButton` hérite de `QAbstractButton`. Et c'est vraiment une info importante, car vous serez peut-être surpris de voir que `QPushButton` contient peu de méthodes qui lui sont propres. C'est normal, une grande partie d'entre elles se trouvent dans sa classe parente `QAbstractButton`.

📌 Il faudra donc absolument consulter aussi `QAbstractButton`, et même sa classe mère `QWidget` (et éventuellement aussi



QObject mais c'est plus rare), si vous voulez connaître toutes les possibilités offertes au final par un QPushButton. Par exemple, `setEnabled` permet d'activer / désactiver le bouton, et cette propriété se trouve dans QWidget.

Les signaux du bouton

Un bouton émet un signal `clicked()` quand on l'active. C'est le signal le plus communément utilisé.

On note aussi les signaux `pressed()` (bouton enfoncé) et `released()` (bouton relâché), mais ils sont plus rares.

Les boutons à 2 états

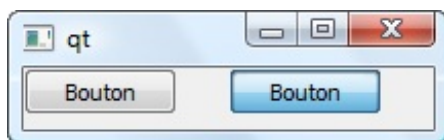
Un bouton peut parfois avoir 2 états : enfoncé et relâché (normal).

Pour activer un bouton à 2 états, utilisez `setCheckable(true)` :

Code : C++

```
QPushButton *bouton = new QPushButton("Bouton", &fenetre);  
bouton->setCheckable(true);
```

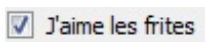
Désormais, un clic sur le bouton le laissera enfoncé, et un nouveau clic le rétablira dans son état normal. Utilisez les signaux `pressed()` et `released()` pour récupérer les changements d'état du bouton.



A gauche un bouton normal, à droite un bouton pressé

QCheckBox : une case à cocher

Une case à cocher `QCheckBox` est généralement associée à un texte de libellé comme ceci :



On définit le libellé de la case lors de l'appel du constructeur :

Code : C++

```
#include <QApplication>  
#include <QWidget>  
#include <QCheckBox>  
  
int main(int argc, char *argv[])  
{  
    QApplication app(argc, argv);  
  
    QWidget fenetre;  
    QCheckBox *checkbox = new QCheckBox("J'aime les frites", &fenetre);  
    fenetre.show();  
}
```

```
        return app.exec();  
    }
```

La case à cocher émet le signal `stateChanged(bool)` lorsqu'on modifie son état. Le booléen en paramètre nous permet de savoir si la case est maintenant cochée ou décochée.

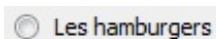
Si vous voulez vérifier à un autre moment si la case est cochée, appelez `isChecked()` qui renvoie un booléen.

On peut aussi faire des cases à cocher à 3 états (le troisième état étant l'état grisé). Renseignez-vous sur la propriété `tristate` pour savoir faire cela. Notez que ce type de case à cocher est relativement rare.

Enfin, sachez que si vous avez plusieurs cases à cocher, vous pouvez les regrouper au sein d'une `QGroupBox`.

QRadioButton : les boutons radio

C'est une case à cocher particulière : une seule case peut être cochée à la fois parmi une liste.



Les radio buttons qui ont le même widget parent sont mutuellement exclusifs. Si vous en cochez un, les autres seront automatiquement décochés.

En général, on place les radio buttons dans une `QGroupBox`. Utiliser des `QGroupBox` différentes vous permet de séparer les groupes de radio buttons.

Voici un exemple d'utilisation d'une `QGroupBox` (qui contient un `layout`, qui contient les `QRadioButton`) :

Code : C++

```
#include <QApplication>  
#include <QtGui>  
  
int main(int argc, char *argv[])  
{  
    QApplication app(argc, argv);  
  
    QWidget fenetre;  
    QGroupBox *groupbox = new QGroupBox("Votre plat préféré",  
    &fenetre);  
  
    QRadioButton *steacks = new QRadioButton("Les steacks");  
    QRadioButton *hamburgers = new QRadioButton("Les hamburgers");  
    QRadioButton *nuggets = new QRadioButton("Les nuggets");  
  
    steacks->setChecked(true);  
  
    QVBoxLayout *vbox = new QVBoxLayout;  
    vbox->addWidget(steacks);  
    vbox->addWidget(hamburgers);  
    vbox->addWidget(nuggets);  
  
    groupbox->setLayout(vbox);  
    groupbox->move(5, 5);  
  
    fenetre.show();  
  
    return app.exec();  
}
```




J'en profite pour signaler que vous pouvez inclure `QtGui` pour automatiquement inclure tous les widgets : `#include <QtGui>`

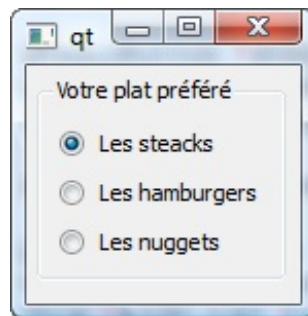
C'est un peu bourrin mais ça marche. 😊

Cela vous évite d'avoir à rajouter un nouveau widget à la liste des includes à chaque fois. Attention par contre, la compilation sera un peu plus longue.

Les radio buttons sont placés dans un layout qui est lui-même placé dans la groupbox, qui est elle-même placée dans la fenêtre. Pfiou ! Le concept des widgets conteneurs est ici utilisé à fond !

Et encore, je n'ai pas fait de layout pour la fenêtre (la flemme, et je ne voulais pas trop encombrer le code), ce qui fait que la taille initiale de la fenêtre est un peu petite, mais ce n'est pas grave c'est pour l'exemple.

Voilà le résultat :



Nota : j'ai une nourriture plus équilibrée que ne le laisse suggérer cette dernière capture d'écran quand même, je vous rassure.



Les afficheurs

Parmi les widgets afficheurs, on compte principalement :

- **QLabel** : le plus important, un widget permettant d'afficher du texte ou une image.
- **QProgressBar** : une barre de progression.

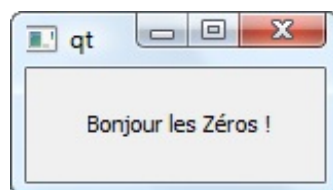
Etudions-les en chœur, sans heurts, dans la joie et la bonne humeur. 😊

QLabel : afficher du texte ou une image

C'est vraiment LE widget de base pour afficher du texte à l'intérieur de la fenêtre. 😊

Nous l'avons déjà utilisé indirectement auparavant, via les cases à cocher ou encore les layouts de formulaire.

Voici un libellé :



... du moins, UN des types de libellés possibles comme nous allons le voir.



QLabel hérite de **QFrame**, qui est un widget de base permettant d'afficher des bordures. Renseignez-vous auprès de **QFrame** pour savoir gérer les différents types de bordure.

Par défaut, un **QLabel** n'a pas de bordure.

Un QLabel peut afficher plusieurs types d'éléments :

- Du texte simple,
- Du texte enrichi (gras, italique, souligné, coloré, avec des liens...),
- Une image,
- Et même une image animée !

Nous allons étudier chacun de ces types de contenu, à l'exception de l'image animée qui sort un peu du cadre du chapitre. Et puis de toute façon, ça ne sert qu'à afficher en pratique des GIF animés, ça nous sera donc peu utile.

Afficher un texte simple

Rien de plus simple, on utilise setText() :

Code : C++

```
label->setText("Bonjour les Zéros !");
```

Mais on peut aussi afficher un texte simple dès l'appel au constructeur comme ceci :

Code : C++

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

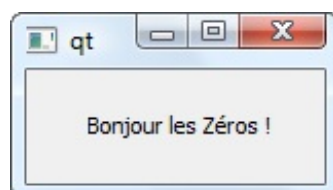
    QWidget fenetre;

    QLabel *label = new QLabel("Bonjour les Zéros !", &fenetre);
    label->move(30, 20);

    fenetre.show();

    return app.exec();
}
```

Le résultat est le même que la capture d'écran que je vous ai montrée plus haut :



Vous pouvez jeter aussi un oeil à la propriété alignment qui permet de définir l'alignement du texte dans le libellé.

Afficher un texte enrichi

Vous pouvez envoyer du texte enrichi (formaté) au QLabel, avec du HTML :

Code : C++

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QLabel *label = new QLabel("Bonjour les <strong>Zéros</strong>
!<br />Etes-vous allés sur le <a
href=\"http://www.siteduzero.com\">Site du Zéro</a> aujourd'hui ?",
&fenetre);
    label->move(30, 20);

    fenetre.show();

    return app.exec();
}
```

Magie, magie, le texte est correctement mis en forme !



C'est beau la technologie quand même. 😊
Et encore, vous n'avez pas tout vu !

Afficher une image

Vous pouvez demander à ce que le QLabel affiche une image.

Comme il n'y a pas de constructeur qui accepte une image en paramètre, on va appeler le constructeur qui prend juste un pointeur vers la fenêtre parente.

Nous demanderons ensuite à ce que le libellé affiche une image à l'aide de setPixmap().

Cette méthode attend un objet de type QPixmap. Après lecture de la doc sur QPixmap, il s'avère que cette classe a un constructeur qui accepte le nom du fichier à charger sous forme de chaîne de caractères.

Nous allons donc afficher notre belle icône de tout à l'heure, mais cette fois en grand et dans la fenêtre :

Code : C++

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
```

```
QLabel *label = new QLabel(&fenetre);  
label->setPixmap(QPixmap("icone.png"));  
label->move(30, 20);  
  
fenetre.show();  
  
return app.exec();  
}
```

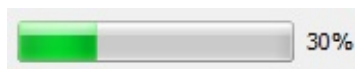
L'icône doit se trouver dans le même dossier que l'exécutable pour que cela fonctionne.

Et voilà le résultat !



QProgressBar : une barre de progression

Les barres de progression sont gérées par `QProgressBar`. Cela permet d'indiquer à l'utilisateur l'avancement des opérations.



Voici quelques propriétés utiles de la barre de progression :

- **maximum** : la valeur maximale que peut prendre la barre de progression.
- **minimum** : la valeur minimale que peut prendre la barre de progression.
- **value** : la valeur actuelle de la barre de progression.

On utilisera donc `setValue` pour changer la valeur de la barre de progression. Par défaut les valeurs sont comprises entre 0 et 100%.



Qt ne peut pas deviner où en sont vos opérations. C'est à vous de calculer le pourcentage d'avancement de vos opérations. La `QProgressBar` se contente juste d'afficher le résultat.

Une `QProgressBar` envoie un signal `valueChanged()` lorsque sa valeur a été modifiée.

A part ça, rien de bien spécial à signaler. Je vous avais déjà fait manipuler les barres de progression dans le chapitre sur les signaux et les slots pour tester la connexion entre les widgets.

Les champs

Nous allons maintenant faire le tour des widgets qui permettent d'entrer des données. C'est la catégorie de widgets la plus importante.

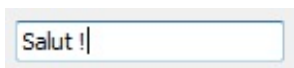
Encore une fois, on ne verra pas tout, mais les principaux d'entre eux :

- **QLineEdit** : champ de texte à une seule ligne.
- **QTextEdit** : champ de texte à plusieurs lignes pouvant afficher du texte mis en forme.
- **QSpinBox** : champ de texte adapté à la saisie de nombre entiers.
- **QDoubleSpinBox** : champ de texte adapté à la saisie de nombre décimaux.
- **QSlider** : un curseur qui permet de sélectionner une valeur.
- **QComboBox** : une liste déroulante.

QLineEdit : champ de texte à une seule ligne

Nous avons utilisé ce widget comme classe d'exemple lors du chapitre sur la lecture de la doc de Qt, vous vous souvenez ?

Un **QLineEdit** est un champ de texte sur une seule ligne :

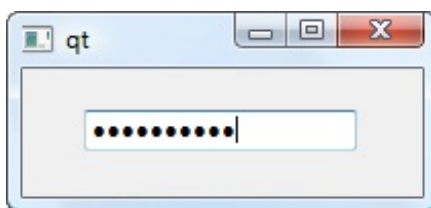


Son utilisation est dans la plupart des cas assez simple. Voici quelques propriétés à connaître :

- **text** : permet de récupérer / modifier le texte contenu dans le champ.
- **alignment** : l'alignement du texte à l'intérieur.
- **echoMode** : type d'affichage du texte. Il faudra utiliser l'énumération EchoMode pour indiquer le type d'affichage. Par défaut les lettres entrées s'affichent, mais on peut aussi faire en sorte que les lettres soient masquées pour les mots de passe.

Code : C++

```
lineEdit->setEchoMode(QLineEdit::Password);
```



- **inputMask** : permet de définir un masque de saisie, pour obliger l'utilisateur à rentrer une chaîne précise (par exemple un numéro de téléphone ne doit pas contenir de lettres). Vous pouvez aussi jeter un oeil aux *validators* qui sont un autre moyen de valider la saisie de l'utilisateur.
- **maxLength** : le nombre de caractères maximum qui peuvent être entrés.
- **readOnly** : le contenu du champ de texte ne peut être modifié. Cette propriété ressemble à enabled (définie dans QWidget), mais avec readOnly on peut quand même copier-coller le contenu du QLineEdit, tandis qu'avec enabled le champ est complètement grisé et on ne peut pas récupérer son contenu.

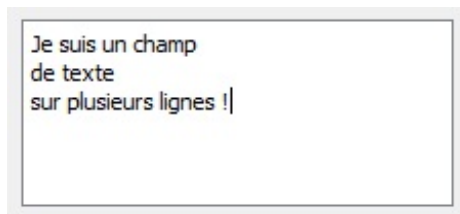
On note aussi plusieurs slots qui permettent de couper / copier / coller / vider / annuler le champ de texte.

Enfin, certains signaux comme `returnPressed()` (l'utilisateur a appuyé sur Entrée) ou `textChanged()` (l'utilisateur a modifié le texte) peuvent être utiles dans certains cas.

QTextEdit : champ de texte à plusieurs lignes

Ce type de champ est similaire à celui qu'on vient de voir, à l'exception du fait qu'il gère l'édition sur plusieurs lignes et, en particulier, qu'il autorise l'affichage de texte enrichi (HTML).

Voici un `QTextEdit` :



Il y a un certains nombre de choses que l'on pourrait voir sur les `QTextEdit` mais ce serait un peu trop long pour ce chapitre qui est plutôt là pour faire une revue rapide des widgets.

Notez les propriétés `plainText` et `html` qui permettent respectivement de récupérer & modifier le contenu sous forme de texte simple et sous forme de texte enrichi en HTML. Tout dépend de l'utilisation que vous en faites, normalement dans la plupart des cas vous utiliserez plutôt `plainText`.



Si vous vous intéressez à l'utilisation du HTML avec Qt, je vous invite à consulter la [liste des balises HTML](#) et [propriétés CSS supportées](#). Vous remarquerez qu'un grand nombre d'éléments sont supportés.

QSpinBox : champ de texte de saisie d'entiers

Une `QSpinBox` est un champ de texte (type `QLineEdit`) qui permet d'entrer uniquement un nombre entier et qui dispose de petits boutons pour augmenter ou diminuer la valeur :



`QSpinBox` hérite de `QAbstractSpinBox` (tiens, encore une classe abstraite !). Vérifiez donc aussi la doc de `QAbstractSpinBox` pour connaître toutes les propriétés de la spinbox.

Voici quelques propriétés intéressantes :

- **accelerated** : permet d'autoriser la spinbox à accélérer la vitesse d'augmentation du nombre si on appuie longtemps sur le bouton.
- **minimum** : valeur minimale que peut prendre la spinbox.
- **maximum** : valeur maximale que peut prendre la spinbox.
- **singleStep** : pas d'incréméntation (par défaut de 1). Si vous voulez que les boutons fassent augmenter la spinbox de 100 en 100, c'est cette propriété qu'il faut modifier !
- **value** : valeur contenue dans la spinbox.
- **prefix** : texte à afficher avant le nombre.
- **suffix** : texte à afficher après le nombre.

QDoubleSpinBox : champ de texte de saisie de nombres décimaux

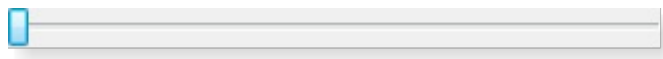
Le `QDoubleSpinBox` est très similaire au `QSpinBox`, à la différence près qu'il travaille sur des nombres décimaux (des double) :



On retrouve la plupart des propriétés de QSpinBox. On peut rajouter la propriété **decimals** qui gère le nombre de chiffres après la virgule affichés par le QDoubleSpinBox.

QSlider : un curseur pour sélectionner une valeur

Un QSlider se présente sous la forme d'un curseur permettant de sélectionner une valeur numérique :



QSlider hérite de QAbstractSlider (damned, encore une classe abstraite !) qui propose déjà un grand nombre de fonctionnalités de base.

Beaucoup de propriétés sont les mêmes que QSpinBox, je ne les relisterai donc pas ici.

Notons la propriété **orientation** qui permet de définir l'orientation du slider (verticale ou horizontale).

Jetez un oeil en particulier à ses signaux, car on connecte en général le signal valueChanged(int) au slot d'autre widget pour répercuter la saisie de l'utilisateur.

Nous avons d'ailleurs manipulé ce widget lors du chapitre sur les signaux et les slots.

QComboBox : une liste déroulante

Une QComboBox est une liste déroulante :



On ajoute des valeurs à la liste déroulante avec la méthode addItem :

Code : C++

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QComboBox *liste = new QComboBox(&fenetre);
    liste->addItem("Paris");
    liste->addItem("Londres");
    liste->addItem("Singapour");
    liste->addItem("Tokyo");
    liste->move(30, 20);

    fenetre.show();

    return app.exec();
}
```

On dispose de propriétés permettant de contrôler le fonctionnement de la QComboBox :

- **count** : nombre d'éléments dans la liste déroulante.
- **currentIndex** : numéro d'indice de l'élément actuellement sélectionné. Les indices commencent à 0. Ainsi, si `currentItem` renvoie 2, c'est que "Singapour" a été sélectionné dans l'exemple précédent.
- **currentText** : texte correspondant à l'élément sélectionné. Si on a sélectionné "Singapour", cette propriété contient donc "Singapour".
- **editable** : indique si le widget autorise l'ajout de valeurs personnalisées ou non. Par défaut, l'ajout de nouvelles valeurs est interdit.
Si le widget est éditable, l'utilisateur pourra rentrer de nouvelles valeurs dans la liste déroulante. Elle se comportera donc aussi comme un champ de texte. L'ajout d'une nouvelle valeur se fait par appui sur la touche "Entrée". Les nouvelles valeurs sont placées par défaut à la fin de la liste.

La QComboBox émet des signaux comme `currentIndexChanged()` qui indique qu'un nouvel élément a été sélectionné et `highlighted()` qui indique l'élément survolé par la souris (ces signaux peuvent envoyer un `int` pour donner l'indice de l'élément ou un `QString` pour le texte).



A noter aussi le widget fils [QFontComboBox](#) qui permet de sélectionner une police parmi une liste proposant une prévisualisation de la police.

Les conteneurs

Normalement, n'importe quel widget peut en contenir d'autres.

Cependant, certains widgets ont été vraiment créés spécialement pour pouvoir en contenir d'autres :

- **QFrame** : un widget pouvant avoir une bordure.
- **QGroupBox** : un widget (que nous avons déjà utilisé) adapté à la gestion des groupes de cases à cocher et de boutons radio.
- **QTabWidget** : un widget gérant plusieurs pages d'onglets.

Nous allons apprendre à les manipuler, en nous intéressant en particulier à celui qui propose des onglets qui est un petit peu délicat. 😊

QFrame : une bordure

[QFrame](#) est très proche de [QWidget](#). En fait, la seule nouveauté c'est qu'il peut générer une bordure. C'est donc un [QWidget](#) basique avec une bordure.



QFrame est une classe de base pour de nombreux widgets qui peuvent avoir une bordure, comme les [QLabel](#). Tout ce que nous allons faire avec les [QFrame](#) ici, tous les widgets qui en héritent peuvent le faire aussi.



Dans la doc de [QFrame](#), regardez au début le paragraphe "Inherited by...". C'est la liste des classes qui héritent de [QFrame](#), et qui disposent donc aussi des fonctionnalités de [QFrame](#).

Un [QFrame](#) possède quelques propriétés pour gérer la forme de la bordure :

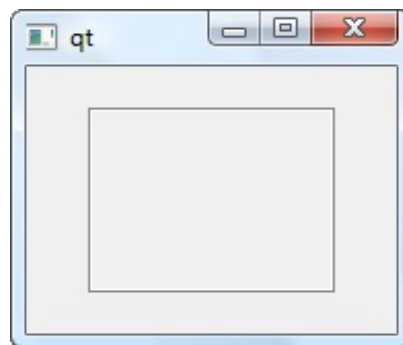
- **frameShape** : le type de bordure. Il faut utiliser une énumération définie par QFrame pour sélectionner la bordure. Consultez la doc pour avoir la liste des types de bordure.
De manière générale je recommande d'utiliser QFrame::StyledPanel (comme sur ma capture d'écran) car cela crée une bordure dans un style adapté à votre OS. Notez aussi QFrame::HLine et QFrame::VLine, un peu particuliers, qui ne créent pas un rectangle mais juste une ligne horizontale ou verticale. Très utile pour séparer les éléments dans sa fenêtre.
- **frameShadow** : l'ombre de la bordure. Par défaut il n'y en a pas, mais vous pouvez définir une ombre qui donne l'impression que le widget est surélevé ou enfoncé.
Regardez les énumérations définies par QFrame pour avoir la liste des possibilités.
- **lineWidth** : l'épaisseur de la ligne de la bordure.
- **midLineWidth** : l'épaisseur de la ligne intermédiaire (utilisé uniquement pour certaines bordure complexes avec une ombre).

Testons la propriété frameShape :

Code : C++

```
QFrame *frame = new QFrame(&fenetre);
frame->setFrameShape(QFrame::StyledPanel);
frame->setGeometry(30, 20, 120, 90);
```

Résultat :



Dans la pratique, le QFrame sert à contenir d'autres widgets (à moins que vous aimiez dessiner des rectangles partout pour le plaisir). Il est donc probable que vous définissiez un layout pour le QFrame et que vous placiez des widgets à l'intérieur. Allez je me fends d'un petit exemple, ça vous rappellera le chapitre sur les layouts :

Code : C++

```
#include <QApplication>
#include <QtGui>

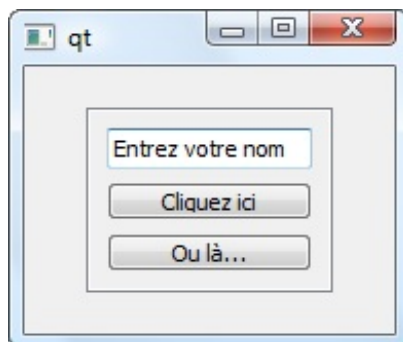
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QFrame *frame = new QFrame(&fenetre);
    frame->setFrameShape(QFrame::StyledPanel);
    frame->setGeometry(30, 20, 120, 90);

    QLineEdit *lineEdit = new QLineEdit("Entrez votre nom");
    QPushButton *bouton1 = new QPushButton("Cliquez ici");
    QPushButton *bouton2 = new QPushButton("Ou là...");
```

```
QVBoxLayout *vbox = new QVBoxLayout;  
vbox->addWidget(lineEdit);  
vbox->addWidget(bouton1);  
vbox->addWidget(bouton2);  
  
frame->setLayout(vbox);  
  
fenetre.show();  
  
return app.exec();  
}
```

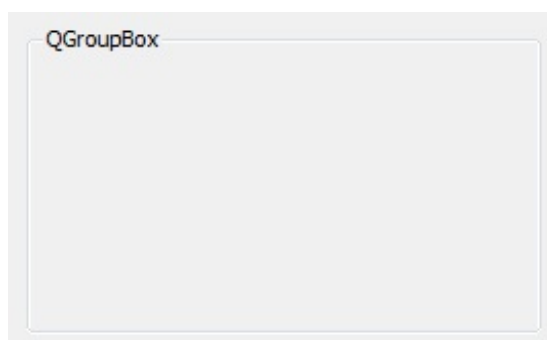


On n'a rien fait de bien nouveau. Le QFrame contient un layout vertical qui organise ses widgets enfants : un QLineEdit et deux QPushButton.

Le QFrame lui-même est placé de manière absolue sur la fenêtre (ses coordonnées sont définies dans setGeometry). Je fais ça pour simplifier l'exemple, mais dans la pratique le QFrame devrait être lui-même placé dans un autre layout (le layout principal de la fenêtre).

QGroupBox : un groupe de widgets

Nous avons déjà utilisé [QGroupBox](#) en pratique plus haut, lorsque nous avons utilisé les boutons radio. Je ne vais donc pas trop m'éterniser dessus.



Un QGroupBox propose de créer une bordure comme QFrame, mais il a en plus l'avantage de permettre de définir un titre pour le conteneur.



[QGroupBox](#) n'hérite pas de QFrame. On n'a donc pas le choix dans le type de bordure, mais on a une propriété **flat** qui permet d'aplatir la bordure.

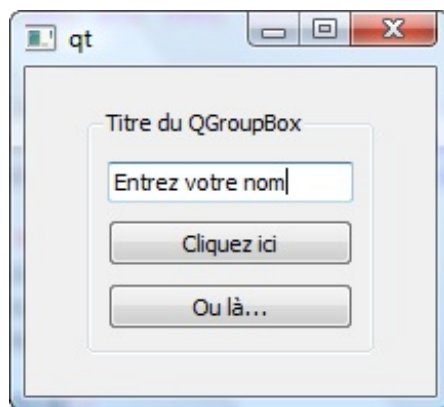
Le titre du conteneur est défini via sa propriété **title**, ou directement lors de l'appel au constructeur :

Code : C++

```
QGroupBox *groupBox = new QGroupBox("Titre du QGroupBox", &fenetre);
```

Exercice : essayez d'adapter l'exemple précédent sur QFrame pour utiliser cette fois un QGroupBox. C'est facile, mais attention aux propriétés spécifiques au QFrame qui ne sont ici plus valables.

Le résultat devrait être le suivant :



Il est aussi possible d'ajouter une case à cocher devant le QGroupBox. Pour cela, mettez sa propriété checkable à true :

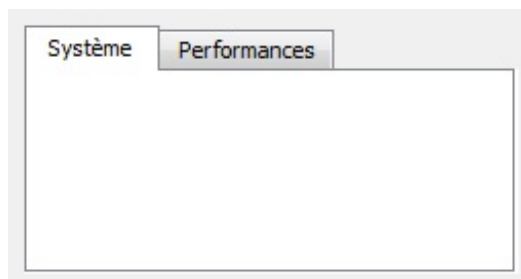
Code : C++

```
groupBox->setCheckable(true);
```

Lorsque la case est cochée, les widgets à l'intérieur sont activés. Lorsqu'elle est décochée, les widgets sont désactivés. Essayez.

QTabWidget : des pages d'onglets

Le **QTabWidget** propose une gestion de plusieurs pages de widgets, organisées sous forme d'onglets :



Ce widget-conteneur est sensiblement plus difficile à utiliser que les autres. En effet, il ne peut contenir qu'un widget par page.



Quoi ? On ne peut pas afficher plus d'un widget par page ???
Mais c'est tout nul !

Sauf... qu'un widget peut en contenir d'autres ! 😊

Et si on utilise un layout pour organiser le contenu de ce widget, on peut arriver rapidement à une super présentation. Le tout est de savoir combiner tout ce qu'on a appris jusqu'ici.

D'après le texte d'introduction de la doc de QTabWidget, ce conteneur doit être utilisé de la façon suivante :

1. Créer un QTabWidget.
2. Créer un QWidget pour chacune des pages (chacun des onglets) du QTabWidget, sans leur indiquer de widget parent.
3. Placer des widgets enfants dans chacun de ces QWidget pour peupler le contenu de chaque page. Utiliser un layout pour positionner les widgets de préférence.
4. Appeler plusieurs fois addTab() pour créer les pages d'onglets en indiquant l'adresse du QWidget qui contient la page à chaque fois.

Bon, c'est un peu plus délicat comme vous pouvez le voir, mais il faut bien un peu de difficulté, ce chapitre était trop facile. 😊

Si on fait tout dans l'ordre, vous allez voir que l'on n'aura pas de problème.

Je vous propose de lire ce code que j'ai créé qui montre un exemple d'utilisation du QTabWidget. Il est un peu long mais il est commenté et vous devriez arriver à le digérer. 😊

Code : C++

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    // 1 : Créer le QTabWidget
    QTabWidget *onglets = new QTabWidget(&fenetre);
    onglets->setGeometry(30, 20, 240, 160);

    // 2 : Créer les pages, en utilisant un widget parent pour
    // contenir chacune des pages
    QWidget *page1 = new QWidget;
    QWidget *page2 = new QWidget;
    QLabel *page3 = new QLabel; // Comme un QLabel est aussi un
    // QWidget (il en hérite), on peut aussi s'en servir de page

    // 3 : Créer le contenu des pages de widgets

    // Page 1

    QLineEdit *lineEdit = new QLineEdit("Entrez votre nom");
    QPushButton *bouton1 = new QPushButton("Cliquez ici");
    QPushButton *bouton2 = new QPushButton("Ou là...");

    QVBoxLayout *vbox1 = new QVBoxLayout;
    vbox1->addWidget(lineEdit);
    vbox1->addWidget(bouton1);
    vbox1->addWidget(bouton2);

    page1->setLayout(vbox1);

    // Page 2

    QProgressBar *progress = new QProgressBar;
    progress->setValue(50);
    QSlider *slider = new QSlider(Qt::Horizontal);
    QPushButton *bouton3 = new QPushButton("Valider");

    QVBoxLayout *vbox2 = new QVBoxLayout;
    vbox2->addWidget(progress);
    vbox2->addWidget(slider);
    vbox2->addWidget(bouton3);
```

```
page2->setLayout (vbox2);

// Page 3 (je ne vais afficher qu'une image ici, pas besoin
de layout)

page3->setPixmap (QPixmap ("icone.png"));
page3->setAlignment (Qt::AlignCenter);

// 4 : ajouter les onglets au QTabWidget, en indiquant la page
qu'ils contiennent
onglets->addTab (page1, "Coordonnées");
onglets->addTab (page2, "Progression");
onglets->addTab (page3, "Image");

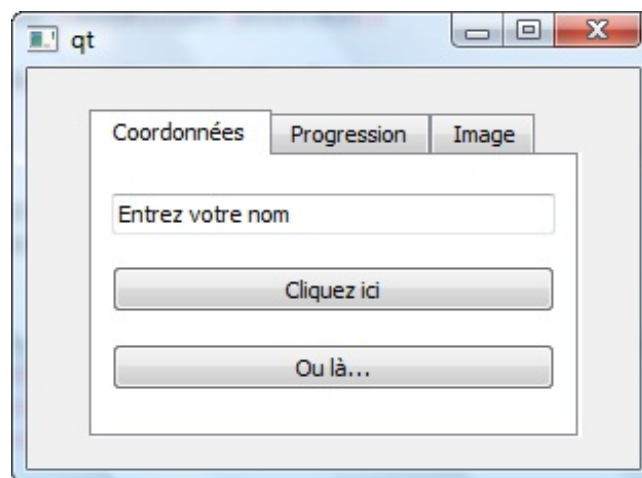
fenetre.show();

return app.exec();
}
```

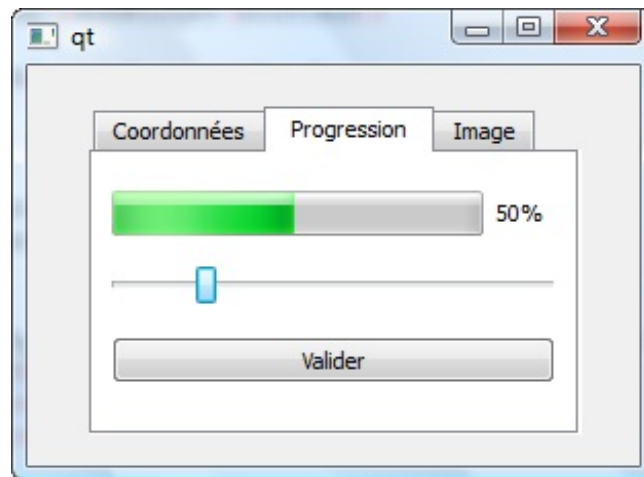
Vous devriez retrouver chacune des étapes que j'ai mentionnées plus haut :

1. Je crée d'abord le QTabWidget que je positionne ici de manière absolue sur la fenêtre (mais je pourrais aussi utiliser un layout).
2. Ensuite, je crée les pages pour chacun de mes onglets. Ces pages sont matérialisées par des QWidget. Vous noterez que pour la dernière page je n'utilise pas un QWidget mais un QLabel. Ca revient au même et c'est compatible car QLabel hérite de QWidget. Sur la dernière page, je me contenterai d'afficher une image.
3. Je crée ensuite le contenu de chacune de ces pages que je dispose à l'aide de layouts verticaux (sauf pour la page 3 qui n'est constituée que d'un widget). Là il n'y a rien de nouveau.
4. Enfin, j'ajoute les onglets avec la méthode addTab(). Je dois indiquer le libellé de l'onglet ainsi qu'un pointeur vers le "widget-page" de cette page.

Résultat, on a un super système d'onglets à 3 pages avec tout plein de widgets dedans !



Page 1



Page 2



Page 3

Je vous conseille de vous entraîner à créer vous aussi un QTabWidget. C'est un bon exercice, et c'est l'occasion de réutiliser la plupart des widgets que l'on a vus dans ce chapitre. 😊

Il faut pratiquer et pratiquer. Ce qu'on fait là ne devrait pas être bien compliqué si vous avez correctement suivi le cours jusqu'ici. Mais il faut quand même pratiquer pour faire des erreurs et se rendre compte qu'on n'avait pas parfaitement tout compris. C'est à partir de ce moment-là seulement que vous commencerez à maîtriser les widgets. 😊

Pfiou !

Il s'agit peut-être du chapitre le plus long que j'aie jamais écrit... mais je tenais à le faire. En fait, c'est un peu paradoxal car tout cela se trouve dans la doc et je vous ai déjà appris à lire la doc, mais j'estimais qu'il devait quand même forcément y avoir un tour d'horizon des principaux widgets dans mon tutoriel, sinon j'aurais eu l'impression d'avoir été incomplet.

Servez-vous de ce chapitre pour vous faire une idée de ce qui existe, mais ensuite je vous conseille très fortement de passer plus de temps sur la doc que sur ce chapitre. En effet, nous sommes loin d'avoir vu toutes les fonctionnalités de ces widgets, de même que nous n'avons pas vu tous les widgets qui existent !

J'ai réservé les widgets les plus complexes pour de futurs chapitres, qui introduiront pour l'occasion un concept de programmation important lorsqu'on programme des GUI : le modèle MVC.

Bon, vous pensez pas qu'il serait temps de pratiquer tout ce qu'on a appris avec un petit TP là ? 😊

TP : ZeroClassGenerator

Je pense que le moment est bien choisi pour vous exercer avec un petit TP. En effet, vous avez déjà vu suffisamment de choses sur Qt pour être en mesure de faire déjà des programmes intéressants.

Quand je me suis dit "*Je vais leur faire faire un TP*", les idées de sujet ne manquaient pas... mais elles étaient toutes un peu "bateau". J'ai pensé à une calculatrice par exemple, et en effet pourquoi pas, mais ce n'était pas très original.

Finalement, après réflexion, j'ai trouvé une idée qui sort un peu de l'ordinaire et qui pourra même vous être utile à vous, programmeurs. 😊

Notre programme s'intitulera le **ZeroClassGenerator**... un programme qui génère le code de base des classes C++ automatiquement en fonction des options que vous choisissez !

Notre objectif

Ne vous laissez pas impressionner par le nom "**ZeroClassGenerator**". Ce TP ne sera pas bien difficile et réutilisera toutes les connaissances que vous avez apprises pour les mettre à profit dans un projet concret.

Ce TP est volontairement modulaire : je vais vous proposer de réaliser un programme de base assez simple, que je vous laisserai coder et que je corrigerai ensuite avec vous. Puis, je vous proposerai un certain nombre d'améliorations intéressantes (non corrigées) pour lesquelles il faudra vous creuser un peu plus les méninges si vous êtes motivés. 😊

Notre ZeroClassGenerator est un programme qui génère le code de base des classes C++. Qu'est-ce que ça veut dire ?

Un générateur de classe C++

Ce programme est un outil graphique qui va créer automatiquement le code source d'une classe en fonction des options que vous aurez choisies.

Vous n'avez jamais remarqué que les classes avaient en général une structure de base similaire qu'il fallait réécrire à chaque fois ? C'est un peu laborieux parfois. Par exemple :

Code : C++

```
#ifndef HEADER_MAGICIEN
#define HEADER_MAGICIEN

class Magicien : public Personnage
{
public:
    Magicien();
    ~Magicien();

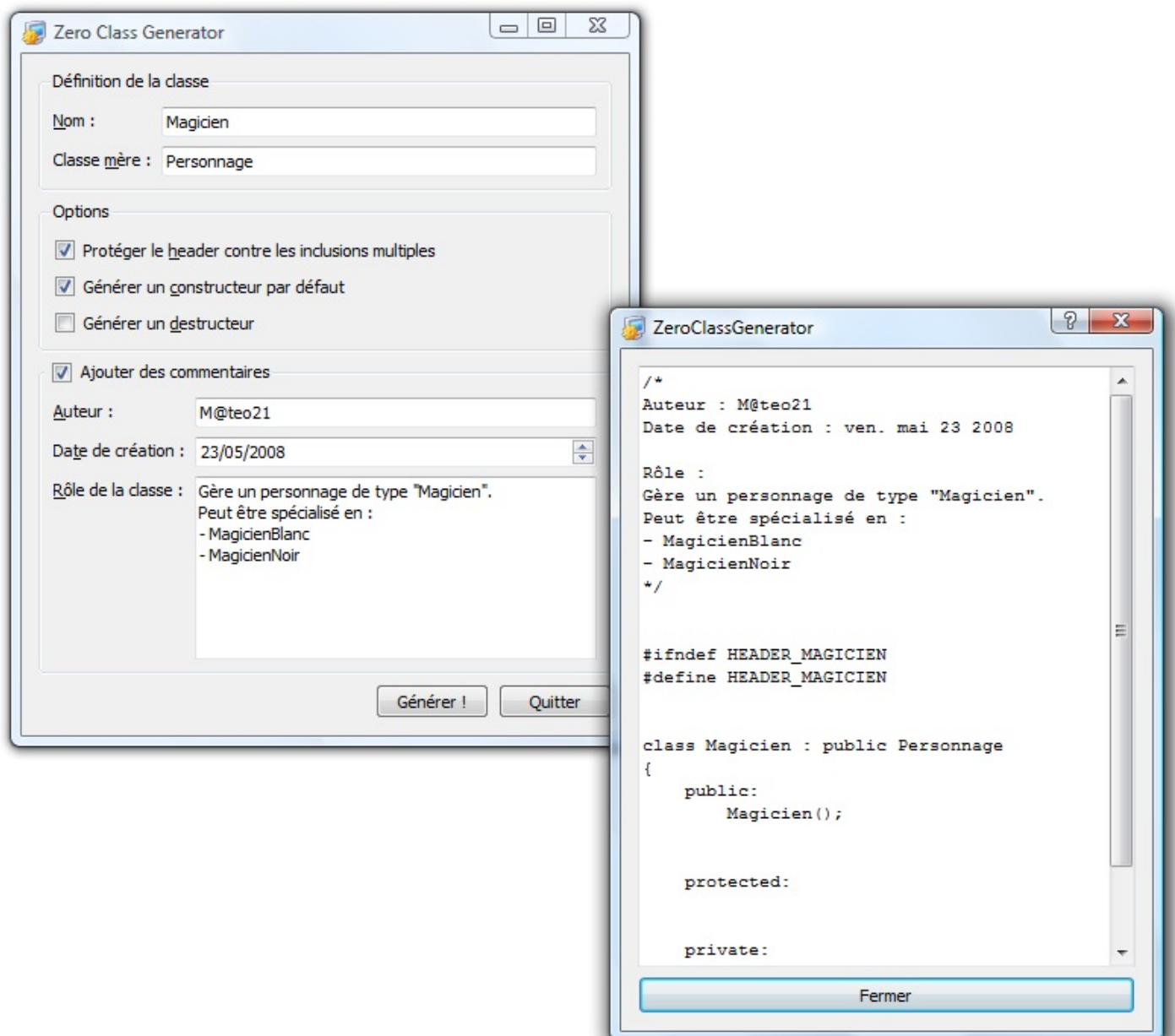
protected:

private:
};

#endif
```

Rien que ça, ça serait bien si on avait un programme capable de générer le squelette de la classe, de définir les portées public, protected et private, de définir un constructeur par défaut et un destructeur, etc.

Nous allons réaliser un GUI (une fenêtre) contenant plusieurs options. Plutôt que de faire une longue liste, je vous propose une capture d'écran du programme final à réaliser :



La fenêtre principale est en haut à gauche, en arrière-plan. L'utilisateur renseigne obligatoirement le champ "Nom", pour indiquer le nom de la classe. Il peut aussi donner le nom de la classe mère.

On propose quelques cases à cocher pour choisir des options comme "Protéger le header contre les inclusions multiples" (la fameuse technique du `#ifndef`, pratique mais un peu lourde à écrire à chaque fois). Il faudra que le nom du `define` soit généré automatiquement à partir du nom de la classe, et mis en majuscules. Pour la mise en majuscules, renseignez-vous auprès de la doc de la classe [QString](#) qui propose plein de choses.

Enfin, on donne la possibilité d'ajouter des commentaires en haut du fichier pour indiquer quel est l'auteur, quelle est la date de création et quel est le rôle de la classe. C'est une bonne habitude en effet que de commenter un peu le début de ses classes pour que l'on ait une idée de ce à quoi elle sert.

Lorsqu'on clique sur le bouton "Générer" en bas, une nouvelle fenêtre s'ouvre (une `QDialog`). Elle affiche le code généré dans un `QTextEdit`, et vous pouvez à partir de là copier/coller ce code dans votre IDE comme `Code::Blocks`.

C'est un début, et je vous proposerai à la fin du chapitre des améliorations intéressantes à ajouter à ce programme. Essayez déjà de réaliser ça correctement, ça représente un peu de travail je peux vous le dire !

Quelques conseils techniques

Avant de vous lâcher tels des fauves dans la jungle, je voudrais vous donner quelques conseils techniques pour vous guider un peu. 😊

Architecture du projet

Je vous recommande de faire une classe par fenêtre. Comme on a 2 fenêtres, et qu'on met toujours le main à part, ça fait 5 fichiers :

- **main.cpp** : contiendra uniquement le main qui ouvre la fenêtre principale (très court).
- **FenPrincipale.h** : header de la fenêtre principale.
- **FenPrincipale.cpp** : l'implémentation des méthodes de la fenêtre principale.
- **FenCodeGenere.h** : le header de la fenêtre secondaire qui affiche le code généré.
- **FenCodeGenere.cpp** : ... et l'implémentation de ses méthodes.

Pour la fenêtre principale, vous pourrez hériter de QWidget comme on l'a toujours fait, ça me semble le meilleur choix. Pour la fenêtre secondaire, je vous conseille d'hériter de QDialog. La fenêtre principale ouvrira la QDialog en appelant sa méthode exec().

La fenêtre principale

Je vous conseille très fortement d'utiliser des layouts. Mon layout principal, si vous regardez bien ma capture d'écran, est un layout vertical. Il contient des QGroupBox.

A l'intérieur des QGroupBox, j'utilise à nouveau des layouts. Je vous laisse le choix du layout qui vous semble le plus adapté à chaque fois.

Pour le QGroupBox "Ajouter des commentaires", il faudra ajouter une case à cocher. Si cette case est cochée, les commentaires seront ajoutés. Sinon, on ne mettra pas de commentaires. Renseignez-vous sur l'utilisation des cases à cocher dans les QGroupBox.



Pour le champ "Date de création", je vous propose d'utiliser un [QDateEdit](#). Ce widget n'a pas été vu dans le chapitre précédent mais je vous fais confiance, il est proche de la [QSpinBox](#) et après lecture de la doc vous devriez savoir vous en servir sans problème.

Vous "dessinerez" le contenu de la fenêtre dans le constructeur de FenPrincipale. Pensez à faire de vos champs de formulaire des attributs de la classe (les QLineEdit, QCheckbox...), afin que toutes les autres méthodes de la classe aient accès à leur valeur.

Lors d'un clic sur le bouton "Générer !", appelez un slot personnalisé. Dans ce slot personnalisé (qui ne sera rien d'autre qu'une méthode de FenPrincipale), vous récupérerez toutes les infos contenues dans les champs de la fenêtre pour générer le code dans une chaîne de caractères (de type QString de préférence).

C'est là qu'il faudra un peu réfléchir sur la génération du code, mais c'est tout à fait faisable. 😊

Une fois le code généré, votre slot appellera la méthode exec() d'un objet de type FenCodeGenere que vous aurez créé pour l'occasion. La fenêtre du code généré s'affichera alors...

La fenêtre du code généré

Beaucoup plus simple, cette fenêtre est constituée d'un QTextEdit et d'un bouton de fermeture.

Pour le QTextEdit, essayez de définir une police à pas fixe (comme "Courier") pour que ça ressemble à du code (parce que le Times New Roman pour rédiger du code c'est moche 😊). Personnellement, j'ai rendu le QTextEdit en mode readOnly pour qu'on ne puisse pas modifier son contenu (juste le copier), mais vous faites comme vous voulez.

Vous connecterez le bouton "Fermer" à un slot spécial de la QDialog qui demande la fermeture et qui indique que tout s'est bien

passé. Je vous laisse trouver dans la doc duquel il s'agit. 🤔



Minute euh... Comment je passe le code généré (de type QString si j'ai bien compris) à la seconde fenêtre de type QDialog ?

Le mieux est de passer cette QString en paramètre du constructeur. Votre fenêtre récupérera ainsi le code et n'aura plus qu'à l'afficher dans son QTextEdit !

Allez hop hop hop, au boulot, à vos éditeurs ! Vous aurez besoin de lire la doc plusieurs fois pour trouver la bonne méthode à appeler à chaque fois, donc n'ayez pas peur d'y aller.

On se retrouve dans la partie suivante pour la... correction !

Correction

Ding !

C'est l'heure de ramasser les copies. 🐱

Bien que je vous aie donné quelques conseils techniques, je vous ai volontairement laissé le choix pour certains petits détails (comme "quelles cases sont cochées par défaut"). Vous pouviez même présenter la fenêtre un peu différemment si vous vouliez. Tout ça pour dire que ma correction n'est pas la correction ultime. Si vous avez fait différemment, ce n'est pas grave. Si vous n'avez pas réussi, ce n'est pas grave non plus, pas de panique : prenez le temps de bien lire mon code et d'essayer de comprendre ce que je fais. Vous devrez être capable par la suite de refaire ce TP sans regarder la correction. 🤔

main.cpp

Comme prévu, ce fichier est tout bête et ne mérite même pas d'explication. 😊

Code : C++

```
#include <QApplication>
#include "FenPrincipale.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Je signale juste qu'on aurait pu charger la langue française comme on l'avait fait dans le chapitre sur les boîtes de dialogue, afin que les menus contextuels et certains boutons automatiques soient traduits en français. Mais c'est du détail, ça ne se verra pas vraiment sur ce projet.

FenPrincipale.h

La fenêtre principale hérite de QWidget comme prévu. Elle utilise la macro Q_OBJECT car nous définissons un slot personnalisé :

Code : C++

```

#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QWidget
{
    Q_OBJECT

public:
    FenPrincipale();

private slots:
    void genererCode();

private:
    QLineEdit *nom;
    QLineEdit *classeMere;
    QCheckBox *protections;
    QCheckBox *genererConstructeur;
    QCheckBox *genererDestructeur;
    QGroupBox *groupCommentaires;
    QLineEdit *auteur;
    QDateEdit *date;
    QTextEdit *role;
    QPushButton *generer;
    QPushButton *quitter;

};

#endif

```

Ce qui est intéressant, ce sont tous les champs de formulaire que j'ai mis en tant qu'attributs (privés) de la classe. Il faudra les initialiser dans le constructeur. L'avantage d'avoir défini les champs en attributs, c'est que toutes les méthodes de la classe y auront accès, et ça nous sera bien utile pour récupérer les valeurs des champs dans notre méthode qui générera le code source.

Notre classe est constituée de 2 méthodes, ce qui est ici largement suffisant :

- **FenPrincipale()** : c'est le constructeur. Il initialisera les champs de la fenêtre, jouera avec les layouts et placera les champs à l'intérieur. Il fera des connexions entre les widgets et indiquera la taille de la fenêtre, son titre, son icône...
- **genererCode()** : c'est une méthode (plus précisément un slot) qui sera connectée au signal "Le bouton Générer a été cliqué". Dès qu'on cliquera sur le bouton, cette méthode sera appelée.
J'ai mis le slot en privé car il n'y a pas de raison qu'une autre classe l'appelle, mais j'aurais aussi bien pu le mettre public.

FenPrincipale.cpp

Bon là c'est le plus gros morceau. Il n'y a que 2 méthodes mais elles sont grosses, ne vous laissez pas impressionner pour autant. Prenez le temps de bien les comprendre. 😊

Code : C++

```

#include "FenPrincipale.h"
#include "FenCodeGenere.h"

FenPrincipale::FenPrincipale()
{
    // Groupe : Définition de la classe

```

```

nom = new QLineEdit;
classeMere = new QLineEdit;

QFormLayout *definitionLayout = new QFormLayout;
definitionLayout->addRow("&Nom :", nom);
definitionLayout->addRow("Classe &mère :", classeMere);

QGroupBox *groupDefinition = new QGroupBox("Définition de la
classe");
groupDefinition->setLayout(definitionLayout);

// Groupe : Options

protections = new QCheckBox("Protéger le &header contre les
inclusions multiples");
protections->setChecked(true);
genererConstructeur = new QCheckBox("Générer un &constructeur
par défaut");
genererDestructeur = new QCheckBox("Générer un &destructeur");

QVBoxLayout *optionsLayout = new QVBoxLayout;
optionsLayout->addWidget(protections);
optionsLayout->addWidget(genererConstructeur);
optionsLayout->addWidget(genererDestructeur);

QGroupBox *groupOptions = new QGroupBox("Options");
groupOptions->setLayout(optionsLayout);

// Groupe : Commentaires

auteur = new QLineEdit;
date = new QDateEdit;
date->setDate(QDate::currentDate());
role = new QTextEdit;

QFormLayout *commentairesLayout = new QFormLayout;
commentairesLayout->addRow("&Auteur :", auteur);
commentairesLayout->addRow("Da&te de création :", date);
commentairesLayout->addRow("&Rôle de la classe :", role);

groupCommentaires = new QGroupBox("Ajouter des commentaires");
groupCommentaires->setCheckable(true);
groupCommentaires->setChecked(false);
groupCommentaires->setLayout(commentairesLayout);

// Layout : boutons du bas (générer, quitter...)
generer = new QPushButton("&Générer !");
quitter = new QPushButton("&Quitter");

QHBoxLayout *boutonsLayout = new QHBoxLayout;
boutonsLayout->setAlignment(Qt::AlignRight);

boutonsLayout->addWidget(generer);
boutonsLayout->addWidget(quitter);

// Définition du layout principal, du titre de la fenêtre, etc.

QVBoxLayout *layoutPrincipal = new QVBoxLayout;
layoutPrincipal->addWidget(groupDefinition);
layoutPrincipal->addWidget(groupOptions);
layoutPrincipal->addWidget(groupCommentaires);
layoutPrincipal->addLayout(boutonsLayout);

setLayout(layoutPrincipal);
setWindowTitle("Zero Class Generator");

```

```

setWindowIcon(QIcon("icone.png"));
resize(400, 450);

// Connexions des signaux et des slots
connect(quitte, SIGNAL(clicked()), qApp, SLOT(quit()));
connect(generer, SIGNAL(clicked()), this, SLOT(genererCode()));
}

void FenPrincipale::genererCode()
{
    // On vérifie que le nom de la classe n'est pas vide, sinon on
    arrête
    if (nom->text().isEmpty())
    {
        QMessageBox::critical(this, "Erreur", "Veuillez entrer au
moins un nom de classe");
        return; // Arrêt de la méthode
    }

    // Si tout va bien, on génère le code
    QString code;

    if (groupCommentaires->isChecked()) // On a demandé à inclure
    les commentaires
    {
        code += "/*\nAuteur : " + auteur->text() + "\n";
        code += "Date de création : " + date->date().toString() +
"\n\n";
        code += "Rôle :\n" + role->toPlainText() + "\n*/\n\n\n";
    }

    if (protections->isChecked())
    {
        code += "#ifndef HEADER_" + nom->text().toUpper() + "\n";
        code += "#define HEADER_" + nom->text().toUpper() +
"\n\n\n";
    }

    code += "class " + nom->text();

    if (!classeMere->text().isEmpty())
    {
        code += " : public " + classeMere->text();
    }

    code += "\n{\n public:\n";
    if (genererConstructeur->isChecked())
    {
        code += " " + nom->text() + " ();\n";
    }
    if (genererDestructeur->isChecked())
    {
        code += " ~" + nom->text() + " ();\n";
    }
    code += "\n\n protected:\n";
    code += "\n\n private:\n";
    code += "};\n\n";

    if (protections->isChecked())
    {
        code += "#endif\n";
    }

    // On crée puis affiche la fenêtre qui affichera le code
    généré, qu'on lui envoie en paramètre
    FenCodeGenere *fenetreCode = new FenCodeGenere(code, this);

```

```
fenetreCode->exec();
}
```



Vous noterez que j'appelle directement la méthode `connect()`, au lieu d'écrire `QWidget::connect()`. En effet, si on est dans une classe qui hérite de `QWidget` (et c'est le cas), on peut se passer de mettre le préfixe `"QWidget::"`.

Pour le constructeur, je pense ne rien avoir à ajouter, je ne fais rien de bien nouveau. Il faut juste être organisé parce qu'il y a pas mal de lignes pour générer la fenêtre.

Par contre, le slot `genererCode` a demandé du travail, même s'il n'est pas si compliqué que ça au final. Il récupère la valeur des champs de la fenêtre (via des méthodes comme `text()` pour les `QLineEdit`). J'ai dû lire la doc plusieurs fois pour chacun de ces widgets afin de savoir comment récupérer le texte, la valeur (si la case est cochée ou pas), etc. Là, c'est juste de la lecture de la doc.

Une `QString` `code` se génère en fonction des choix que vous avez fait.

Une erreur se produit et la méthode s'arrête s'il n'y a pas au moins un nom de classe défini.

Tout à la fin de `genererCode()`, on n'a plus qu'à appeler la fenêtre secondaire et à lui envoyer le code généré :

Code : C++

```
FenCodeGenere *fenetreCode = new FenCodeGenere(code, this);
fenetreCode->exec();
```

Le code est envoyé lors de la construction de l'objet. La fenêtre sera affichée lors de l'appel à `exec()`.

FenCodeGenere.h

La fenêtre du code généré est beaucoup plus simple que sa parente :

Code : C++

```
#ifndef HEADER_FENCODEGENERE
#define HEADER_FENCODEGENERE

#include <QtGui>

class FenCodeGenere : public QDialog
{
public:
    FenCodeGenere(QString &code, QWidget *parent);

private:
    QTextEdit *codeGenere;
    QPushButton *fermer;
};

#endif
```

Il y a juste un constructeur et deux petits widgets de rien du tout. 😊

FenCodeGenere.cpp

Le constructeur prend 2 paramètres :

- Une référence vers la QString qui contient le code.
- Un pointeur vers la fenêtre parente.

Code : C++

```
#include "FenCodeGenere.h"

FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0) :
    QDialog(parent)
{
    codeGenere = new QTextEdit();
    codeGenere->setPlainText(code);
    codeGenere->setReadOnly(true);
    codeGenere->setFont(QFont("Courier"));
    codeGenere->setLineWrapMode(QTextEdit::NoWrap);

    fermer = new QPushButton("Fermer");

    QVBoxLayout *layoutPrincipale = new QVBoxLayout;
    layoutPrincipale->addWidget(codeGenere);
    layoutPrincipale->addWidget(fermer);

    resize(350, 450);
    setLayout(layoutPrincipale);

    connect(fermer, SIGNAL(clicked()), this, SLOT(accept()));
}
```

C'est un rappel, mais je pense qu'il ne fera pas de mal : le paramètre *parent* est transféré au constructeur de la classe-mère QDialog dans cette ligne :

Code : C++

```
FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0) :
    QDialog(parent)
```

Schématiquement, le transfert se fait comme ceci :

```
FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0) : QDialog(parent)
```



Je pense que s'il y avait juste ça vous comprendrez tous :

Code : C++

```
FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0)
```

La nouveauté (enfin, on en a parlé dans le chapitre sur l'héritage quand même 🤔), c'est qu'on appelle aussi le constructeur de la classe-mère QDialog et on lui transfère le paramètre parent avec le code : QDialog(parent)



Pourquoi avoir appelé le constructeur de QDialog et pourquoi lui avoir envoyé en paramètre un pointeur vers la fenêtre mère ?

Même si ce n'est pas obligatoire en fait, il est conseillé lors de la création d'une QDialog d'indiquer quelle est la fenêtre mère. La QDialog se centre automatiquement par rapport à la fenêtre mère, entre autres choses.

Télécharger le projet

Vous pouvez aussi télécharger le projet zippé :

[Télécharger le projet ZeroClassGenerator \(25 Ko\)](#)

Ce zip contient :

- Les fichiers source .cpp et .h
- Le projet .cbp pour ceux qui utilisent Code::Blocks
- L'exécutable Windows et son icône. Attention, il faudra mettre les DLL de Qt dans le même dossier si vous voulez que le programme puisse s'exécuter.

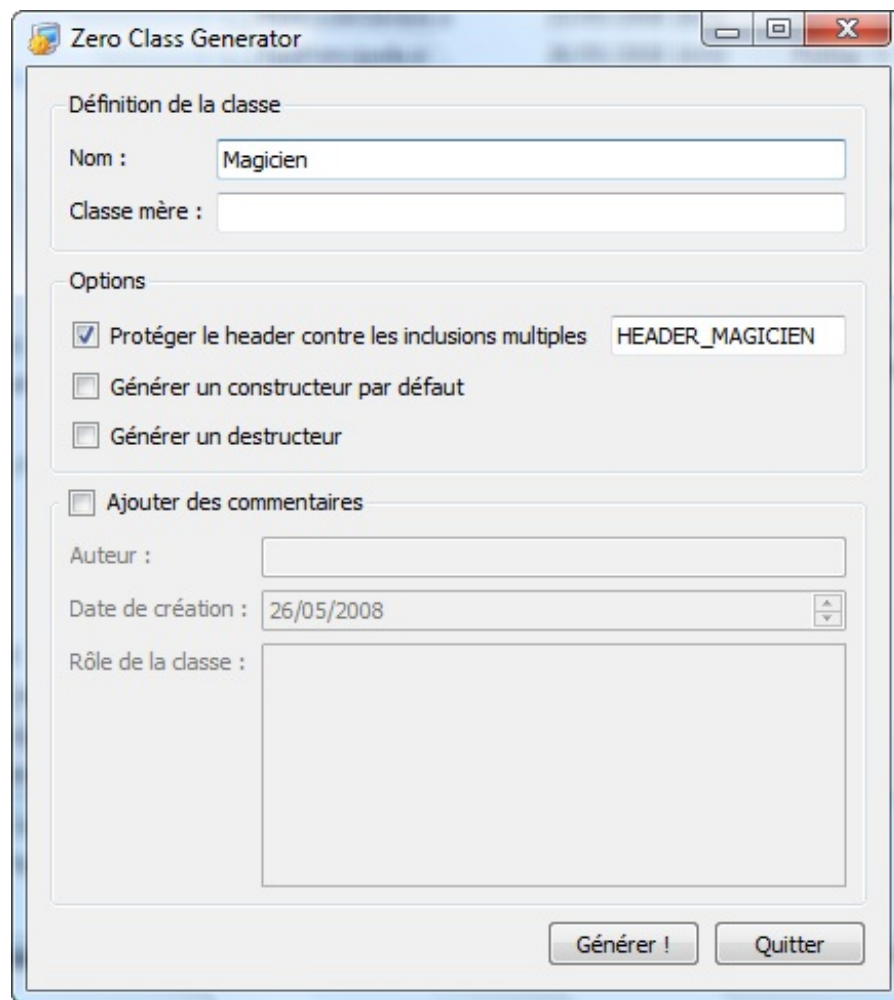
Des idées d'améliorations

Vous pensiez en avoir fini ?

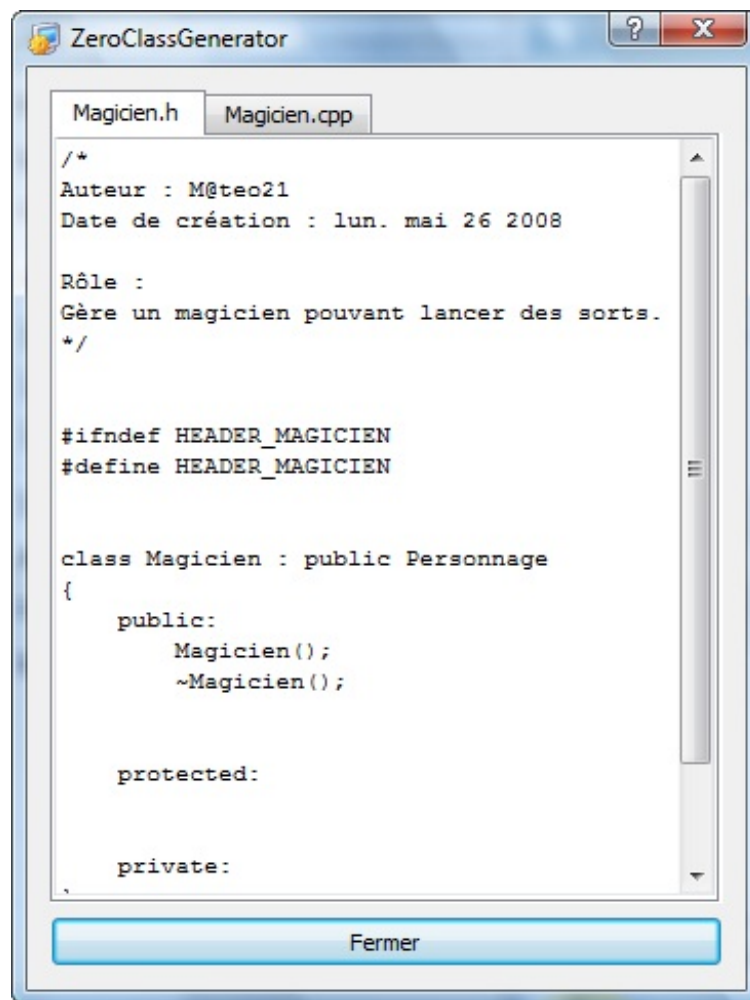
Que nenni ! Un tel TP n'attend qu'une seule chose : être amélioré !

Voici une liste de suggestions qui me passent par la tête pour améliorer le ZeroCodeGenerator, mais vous pouvez inventer les vôtres :

- Lorsqu'on coche "Protéger le header contre les inclusions multiples", un define (aussi appelé "header guard") est généré. Par défaut, ce header guard est de la forme HEADER_NOMCLASSE. Pourquoi ne pas l'afficher en temps réel dans un libellé lorsqu'on tape le nom de la classe ? Ou, mieux, affichez-le en temps réel dans un QLineEdit pour que la personne puisse le modifier si elle le désire.
Le but est de vous faire travailler les signaux et les slots.



- Ajoutez d'autres options de génération de code. Par exemple, vous pouvez proposer d'inclure le texte légal d'une licence libre (comme la GPL) dans les commentaires d'en-tête si la personne fait un logiciel libre, vous pouvez demander quels headers inclure, la liste des attributs, générer automatiquement les accesseurs pour ces attributs, etc. Attention, il faudra peut-être utiliser des widgets de liste un peu plus complexes, comme le [QListWidget](#). Je ne vous l'ai pas encore présenté, mais rien ne vous interdit de prendre de l'avance. 😊
- Pour le moment on ne génère que le code du fichier .h. Même s'il y a moins de travail, ça serait bien de générer aussi le .cpp. Je vous propose d'utiliser un [QTabWidget](#) (des onglets) pour afficher le code .h et le .cpp dans la boîte de dialogue du code généré.



- On ne peut que voir et copier / coller le code généré. C'est bien, mais comme vous je pense que si on pouvait enregistrer le résultat dans des fichiers ce serait du temps de gagné pour l'utilisateur. Je vous propose d'ajouter dans la QDialog un bouton pour sauvegarder dans des fichiers.
Ce bouton ouvrira une fenêtre qui demandera dans quel dossier enregistrer les fichiers .h et .cpp. Le nom de ces fichiers sera automatiquement généré en fonction du nom de la classe.
Pour l'enregistrement dans des fichiers, regardez du côté de la classe [QFile](#). Bon courage. 😊
- C'est un détail, mais les menus contextuels (quand on fait un clic droit sur un champ de texte par exemple) sont en anglais. Je vous avais parlé dans un des chapitres précédents d'une technique permettant de les avoir en français, un code à placer au début du main(). Je vous laisse le retrouver !
- On vérifie si le nom de la classe n'est pas vide, mais on ne vérifie pas s'il contient des caractères invalides (comme un espace, des accents, des guillemets...). Il faudrait afficher une erreur si le nom de la classe n'est pas valide.
Pour valider le texte saisi, vous avez 2 techniques : utiliser un `inputMask()`, ou un `validator()`. L'`inputMask` est peut-être le plus simple des deux, mais ça vaut le coup d'avoir pratiqué les deux. Pour savoir faire ça, direction la [doc de QLineEdit](#).

Voilà pour un petit début d'idées d'améliorations. Il y a déjà de quoi faire pour que vous ne puissiez pas dormir pendant quelques nuits, gnark gnark gnark. 😊

Comme toujours pour les TP, si vous êtes bloqués rendez-vous sur les forums du Site du Zéro pour demander de l'aide. Bon courage à tous !

J'espère que le sujet de ce TP vous a plu, j'ai essayé de trouver quelque chose d'original et d'éviter les sujets habituels comme "Faire une calculatrice".

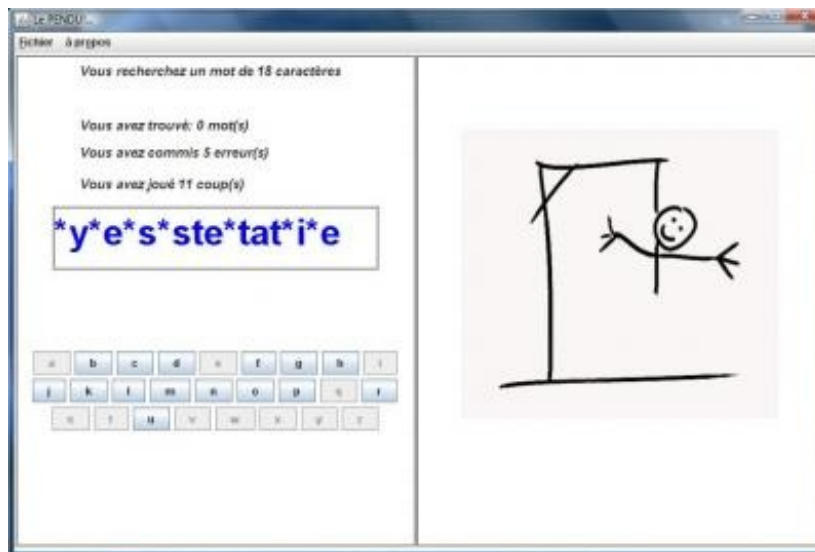
Ceci étant, si vous avez envie de faire une calculatrice, surtout foncez ! Plus vous pratiquez, plus vous progressez (même si vous êtes parfois confrontés à des difficultés). C'est une règle immuable.

Parmi les TP sur lesquels vous pourriez vous entraîner à votre niveau, je vous suggère de vous inspirer d'autres TP des cours du site (parfois réalisés dans d'autres langages), et d'essayer de construire une interface graphique avec Qt :

- [Le jeu du Plus ou Moins](#) : faites deviner un nombre à l'utilisateur. Ce devrait être assez simple à réaliser, et ça vous fera pratiquer un peu plus.
- [Le jeu du Pendu](#) : un peu plus complexe mais toujours très intéressant, le Pendu sous forme de GUI peut être un très bon

exercice.

Ci-dessous, un exemple de jeu de pendu sous forme de GUI, ici réalisé en Java par cysboy [dans son tutoriel](#) (même si c'est du Java, ça ne change rien, c'est pour vous donner une idée de l'interface à réaliser) :



Amusez-vous bien. 😊

La fenêtre principale

Intéressons-nous maintenant à la fenêtre principale de vos applications.

Pour le moment, nous avons créé des fenêtres plutôt basiques en héritant de `QWidget`. C'est en effet largement suffisant pour de petites applications, mais au bout d'un moment on a besoin de plus d'outils.

La classe `QMainWindow` a été spécialement créée pour gérer la fenêtre principale de votre application quand celle-ci est complexe. Parmi les fonctionnalités offertes par `QMainWindow`, on trouve :

- Les menus
- La barre d'outils
- Les docks
- La barre d'état

A la fin de ce chapitre, vous pourrez vraiment faire tout ce que vous voulez de votre fenêtre principale !

Présentation de `QMainWindow`

La classe `QMainWindow` hérite directement de `QWidget`. C'est un widget généralement utilisé une seule fois par programme, et qui sert uniquement à créer la fenêtre principale de l'application.

Certaines applications simples n'ont pas besoin de recourir à la `QMainWindow`. On va supposer ici que vous vous attaquez à un programme complexe et d'envergure. 😊

Structure de la `QMainWindow`

Avant toute chose, il me semble indispensable de vous présenter l'organisation d'une `QMainWindow`. Commençons par analyser le schéma ci-dessous :

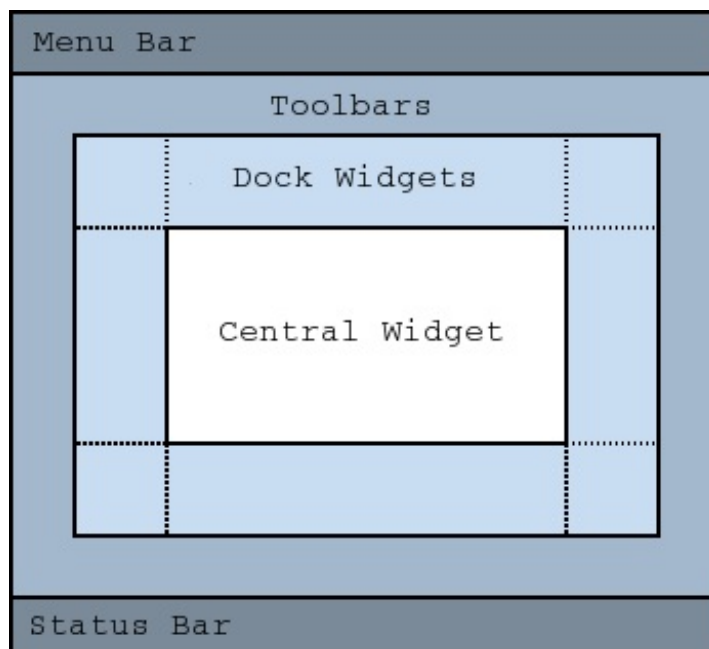


Schéma honteusement recopié de la doc de `QMainWindow` 😊

Une fenêtre principale peut être constituée de tout cela. Et j'ai bien dit *peut*, car rien ne vous oblige à utiliser à chaque fois chacun de ces éléments.

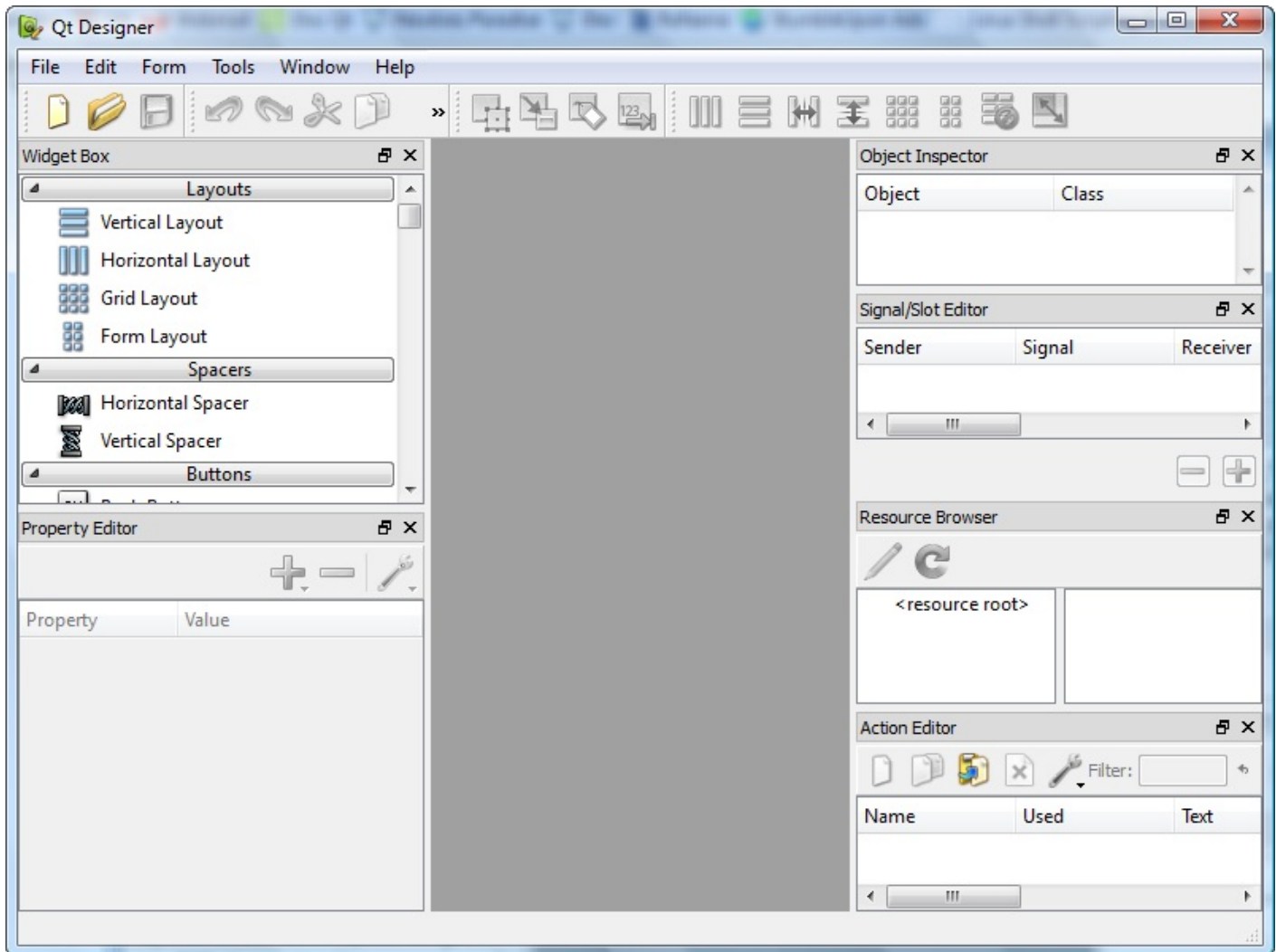
Détaillons les éléments :

- **Menu Bar** : c'est la barre de menus. C'est là que vous allez pouvoir créer votre menu Fichier, Edition, Affichage, Aide, etc.
- **Toolbars** : les barres d'outils. Dans un éditeur de texte, on a par exemple des icônes pour créer un nouveau fichier, pour enregistrer, etc.

- **Dock Widgets** : plus complexes et plus rarement utilisés, ces docks sont des conteneurs que l'on place autour de la fenêtre principale. Ils peuvent contenir des outils, par exemple les différents types de pinceaux que l'on peut utiliser quand on fait un logiciel de dessin.
- **Central Widget** : c'est le coeur de la fenêtre, là où il y aura le contenu proprement dit.
- **Status Bar** : la barre d'état. Elle affiche en général l'état du programme (Prêt / Enregistrement en cours, etc.).

Exemple de QMainWindow

Pour imaginer ces éléments en pratique, je vous propose de prendre pour exemple le programme Qt Designer :



Vous repérez en haut la **barre de menus** : File, Edit, Form...

En dessous, on a la **barre d'outils**, avec les icônes pour créer un nouveau projet, ouvrir un projet, enregistrer, annuler...

Autour (sur la gauche et la droite), on a les fameux **docks**. Ils servent ici à sélectionner le widget que l'on veut utiliser, ou à éditer les propriétés du widget par exemple.

Au centre, dans la partie grise où il n'y a rien, c'est la **zone centrale**. Lorsqu'un document est ouvert, cette zone l'affiche. La zone centrale peut afficher un ou plusieurs documents à la fois, comme on le verra plus loin.

Enfin, en bas il y a normalement la **barre de statut**, mais Qt Designer n'en utilise pas vraiment visiblement (en tout cas rien n'est affiché en bas).

Je vous propose de passer en revue chacune de ces sections dans ce chapitre. Nous commencerons par parler du "**Central Widget**", car c'est quand même lui le plus important et c'est le seul véritablement indispensable. 🤖

Le code de base

Pour suivre ce chapitre, il va falloir créer un projet en même temps que moi. Nous allons créer notre propre classe de fenêtre principale qui héritera de QMainWindow, car c'est comme cela qu'on fait dans 99,99 % des cas.

Notre projet contiendra 3 fichiers :

- **main.cpp** : la fonction main().
- **FenPrincipale.h** : définition de notre classe FenPrincipale, qui héritera de QMainWindow.
- **FenPrincipale.cpp** : implémentation des méthodes de la fenêtre principale.

main.cpp

Code : C++

```
#include <QApplication>
#include <QtGui>
#include "FenPrincipale.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

FenPrincipale.h

Code : C++

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QMainWindow
{
public:
    FenPrincipale();

private:
};

#endif
```

FenPrincipale.cpp

Code : C++

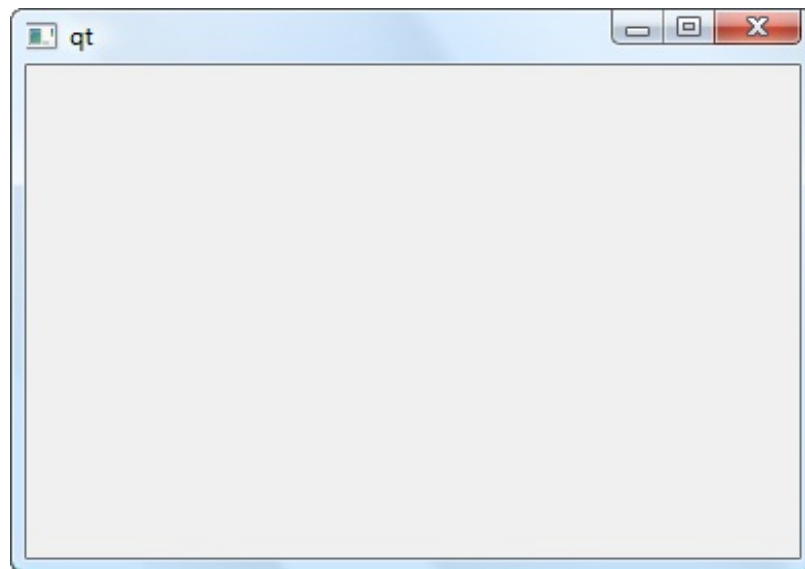
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
}

```

Résultat

Si tout va bien, ce code devrait avoir pour effet d'afficher une fenêtre vide, toute bête :



Si c'est ce qui s'affiche chez vous, c'est bon, nous pouvons commencer. 😊

La zone centrale (SDI et MDI)

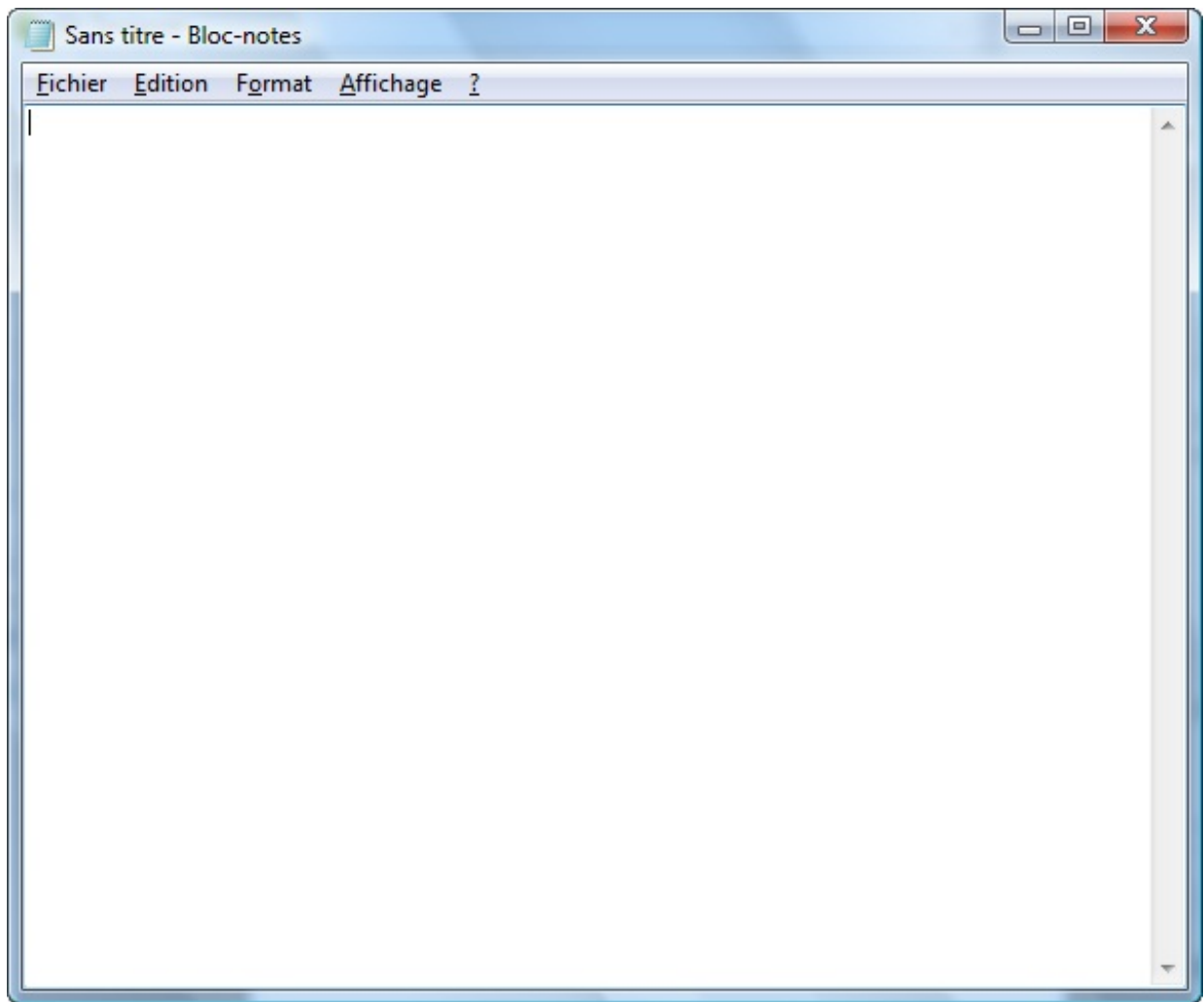
La zone centrale de la fenêtre principale est prévue pour contenir un et un seul widget.

C'est comme pour les onglets. On y insère un QWidget (ou une de ses classes filles) et on s'en sert comme conteneur pour mettre d'autres widgets à l'intérieur si besoin est.

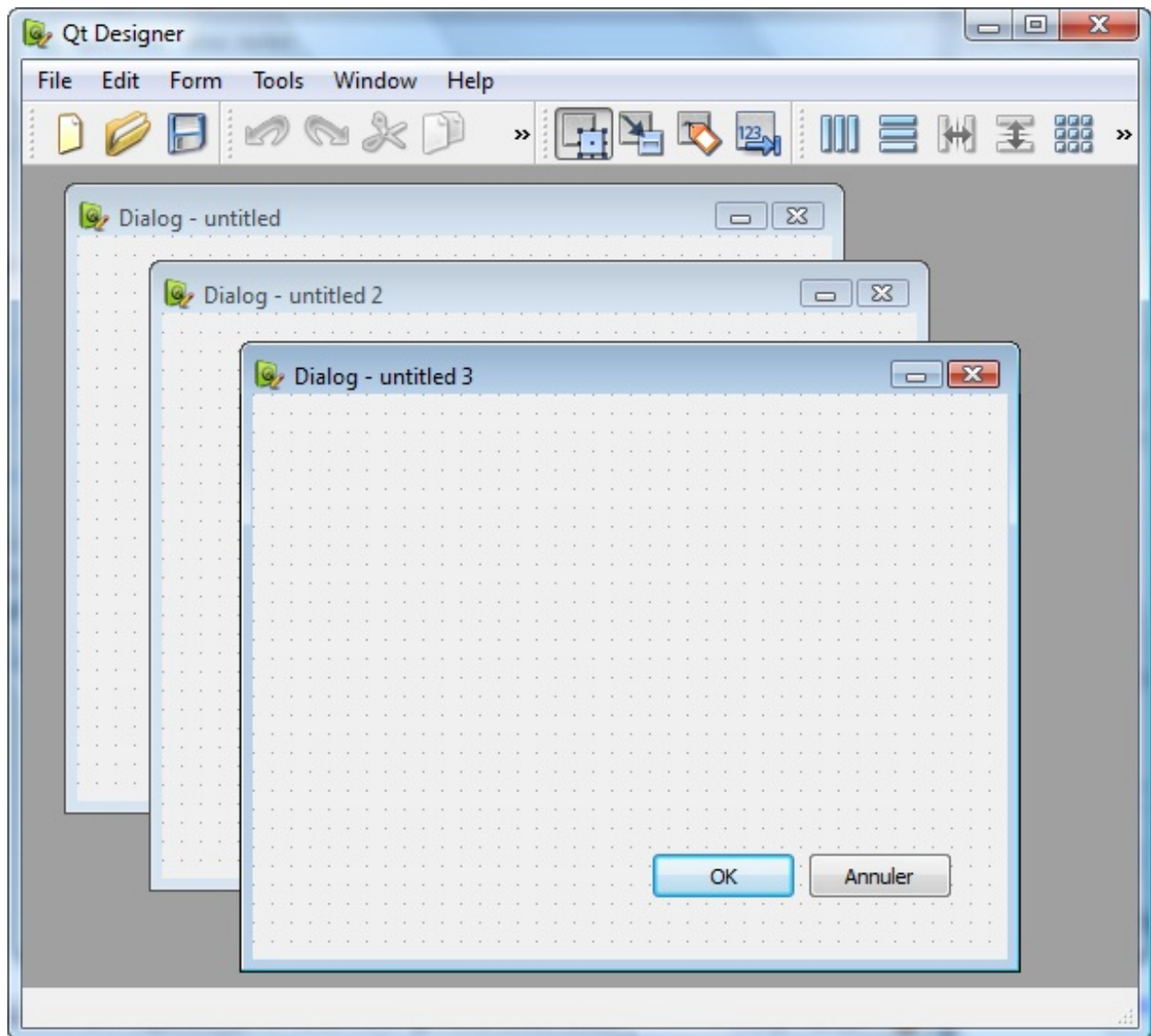
Nous allons refaire la manipulation ici pour s'assurer que tout le monde comprenne comment cela fonctionne.

Sachez tout d'abord qu'on distingue 2 types de QMainWindow :

- **Les SDI** (Single Document Interface) : elles ne peuvent afficher qu'un document à la fois. C'est le cas de Bloc-Notes par exemple :



- **Les MDI (Multiple Document Interface)** : elles peuvent afficher plusieurs documents à la fois. Elles affichent des sous-fenêtres dans la zone centrale. C'est le cas par exemple de Qt Designer :



Définition de la zone centrale (type SDI)

On utilise la méthode `setCentralWidget()` de la `QMainWindow` pour indiquer quel widget contiendra la zone centrale. Faisons cela dans le constructeur de `FenPrincipale` :

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QWidget *zoneCentrale = new QWidget;
    setCentralWidget(zoneCentrale);
}
```

Visuellement, ça ne change rien pour le moment. Par contre ce qui est intéressant, c'est qu'on a maintenant un `QWidget` qui sert de conteneur pour les autres widgets de la zone centrale de la fenêtre.

On peut donc y insérer des widgets au milieu :

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QWidget *zoneCentrale = new QWidget;

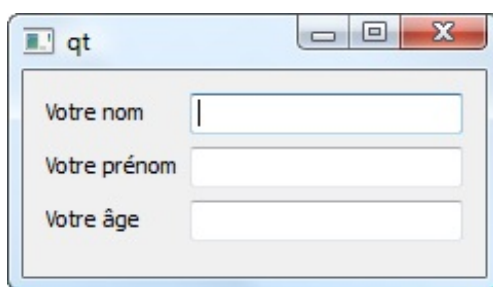
    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;

    QFormLayout *layout = new QFormLayout;
    layout->addRow("Votre nom", nom);
    layout->addRow("Votre prénom", prenom);
    layout->addRow("Votre âge", age);

    zoneCentrale->setLayout(layout);

    setCentralWidget(zoneCentrale);
}
```

Vous noterez que j'ai repris le code du chapitre sur les layouts. J'ai un poil dans la main aujourd'hui, pas envie d'inventer de nouveaux exemples surtout que ça fait exactement la même chose. 🤔



Bon, je reconnais qu'on ne fait rien de bien excitant pour le moment. Mais maintenant vous savez au moins comment définir un widget central pour une QMainWindow, et ça mine de rien c'est important. 😊

Définition de la zone centrale (type MDI)

Les choses se compliquent un peu (mais pas trop 🤔) si vous voulez créer un programme MDI... par exemple un éditeur de texte qui peut gérer plusieurs documents à la fois.

Nous allons utiliser pour cela une [QMdiArea](#), qui est une sorte de gros widget conteneur capable d'afficher plusieurs sous-fenêtres.

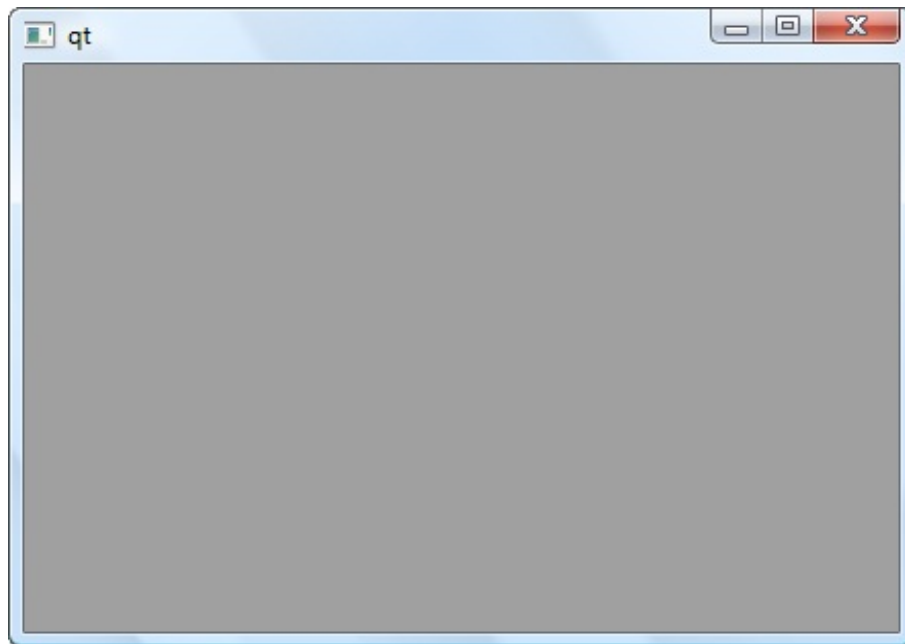
On peut se servir du QMdiArea comme de widget conteneur pour la zone centrale :

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMdiArea *zoneCentrale = new QMdiArea;
    setCentralWidget(zoneCentrale);
}
```

La fenêtre est maintenant prête à accepter des sous-fenêtres :



Si le fond gris par défaut ne vous plaît pas, vous pouvez changer le fond (en mettant une autre couleur ou une image) en appelant `setBackground()`.

On crée ces sous-fenêtres en appelant la méthode `addSubWindow()` du `QMdiArea`. Cette méthode attend en paramètre le widget que la sous-fenêtre doit afficher à l'intérieur.

Là encore, vous pouvez créer un `QWidget` générique qui contiendra d'autres widgets, eux-mêmes organisés selon un layout.

On va faire plus simple dans notre exemple : on va faire en sorte que les sous-fenêtres contiennent juste un `QTextEdit` (pour notre éditeur de texte) :

Code : C++

```
#include "FenPrincipale.h"

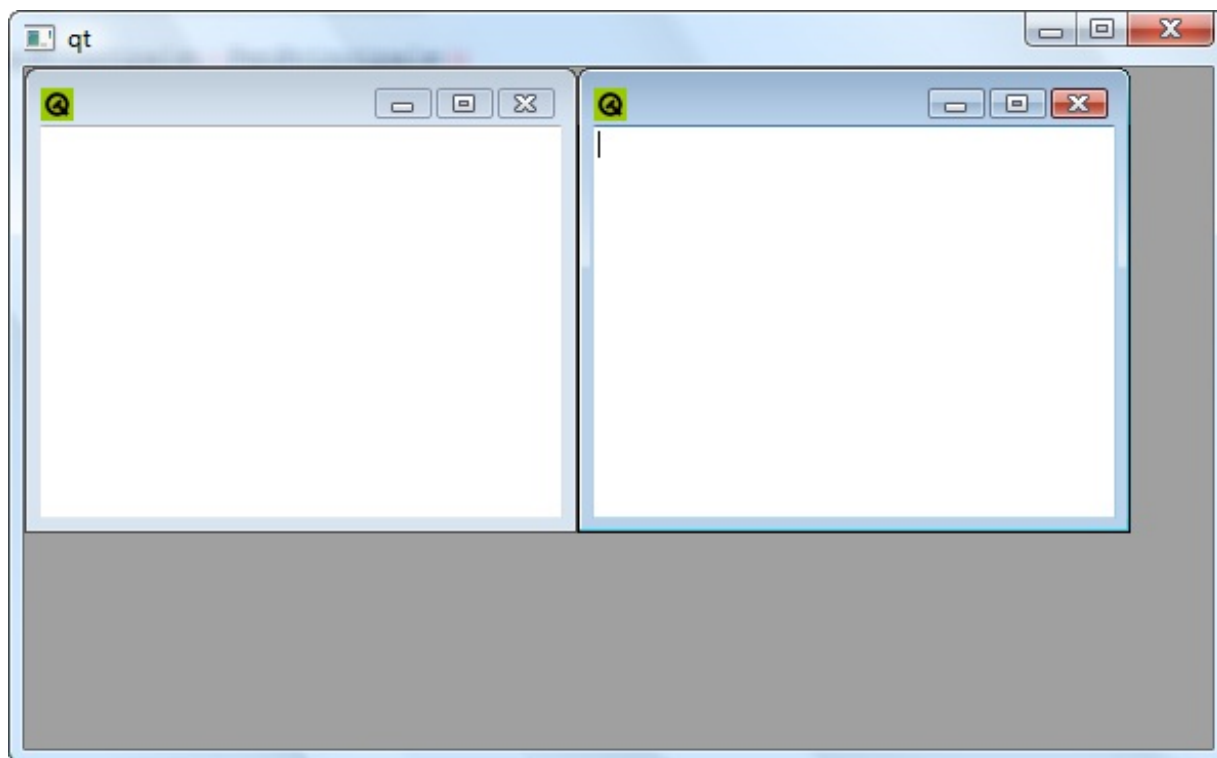
FenPrincipale::FenPrincipale()
{
    QMdiArea *zoneCentrale = new QMdiArea;

    QTextEdit *zoneTexte1 = new QTextEdit;
    QTextEdit *zoneTexte2 = new QTextEdit;

    QMdiSubWindow *sousFenetre1 = zoneCentrale-
>addSubWindow(zoneTexte1);
    QMdiSubWindow *sousFenetre2 = zoneCentrale-
>addSubWindow(zoneTexte2);

    setCentralWidget(zoneCentrale);
}
```

Résultat, on a une fenêtre principale qui contient plusieurs sous-fenêtres à l'intérieur :



Ces fenêtres peuvent être réduites ou agrandies à l'intérieur même de la fenêtre principale. On peut leur définir un titre et une icône avec les bonnes vieilles méthodes `setWindowTitle`, `setWindowIcon`, etc.

C'est quand même dingue tout ce qu'on peut faire en quelques lignes de code avec Qt ! 🤖

Sur cet exemple, j'ai créé des "fenêtres-zones_de_texte", un peu comme j'avais fait des "fenêtres-boutons" dans les premiers chapitres sur Qt.



Bien sûr, dans la pratique, les sous-fenêtres seront peut-être un peu plus complexes. On n'utilisera pas un `QTextEdit` directement mais plutôt un `QWidget` qui contiendra un layout qui contiendra des widgets. Bref, vous m'avez compris.



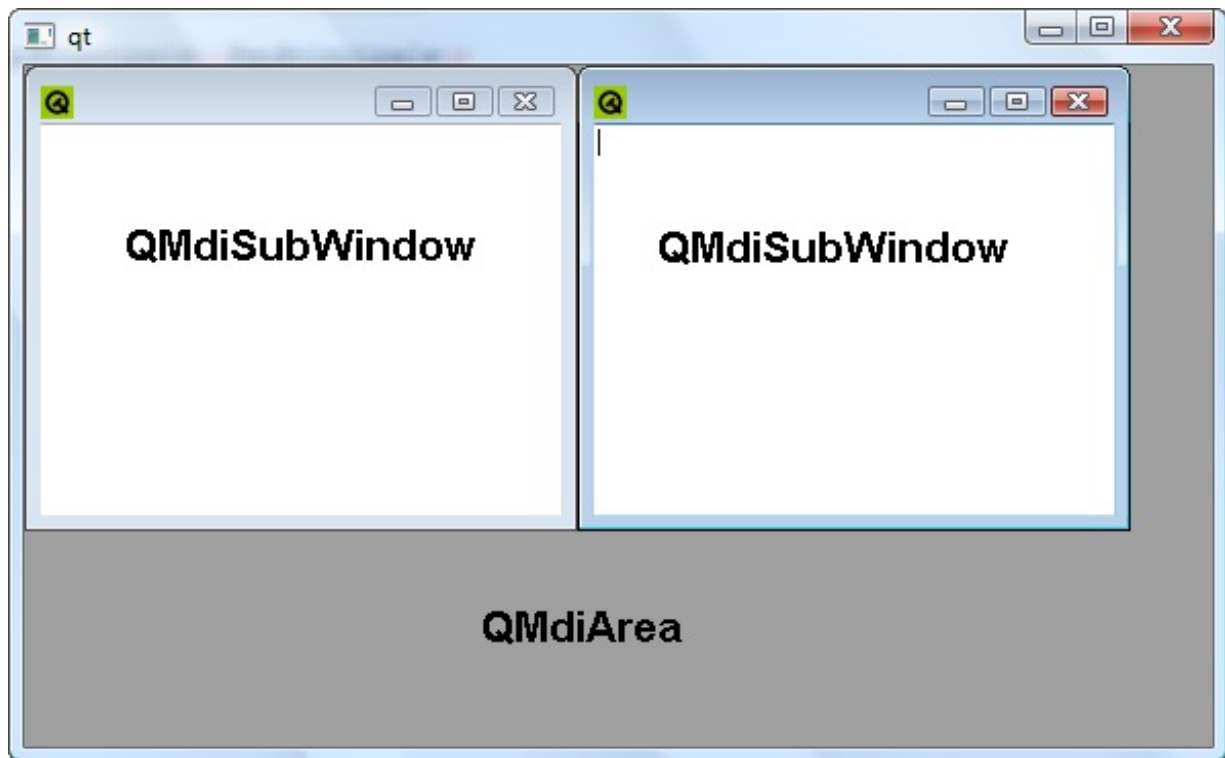
Vous remarquerez que `addSubWindow()` renvoie un pointeur sur une `QMdiSubWindow` : ce pointeur représente la sous-fenêtre qui a été créée. Ça peut être une bonne idée de garder ce pointeur pour la suite. Vous pourrez ainsi supprimer la fenêtre en appelant `removeSubWindow()`.

Sinon, sachez que vous pouvez retrouver à tout moment la liste des sous-fenêtres créées en appelant `subWindowList()`. Cette méthode renvoie la liste des `QMdiSubWindow` contenues dans la `QMdiArea`.

...

Ça va vous y retrouvez ? 🤖

Allez un petit schéma pour être sûr que vous avez compris :



On a donc une zone centrale `QMdiArea` qui contient plusieurs sous-fenêtres, représentées par des `QMdiSubWindow`. Chacune de ces sous-fenêtres est indépendante et peut contenir ses propres widgets.

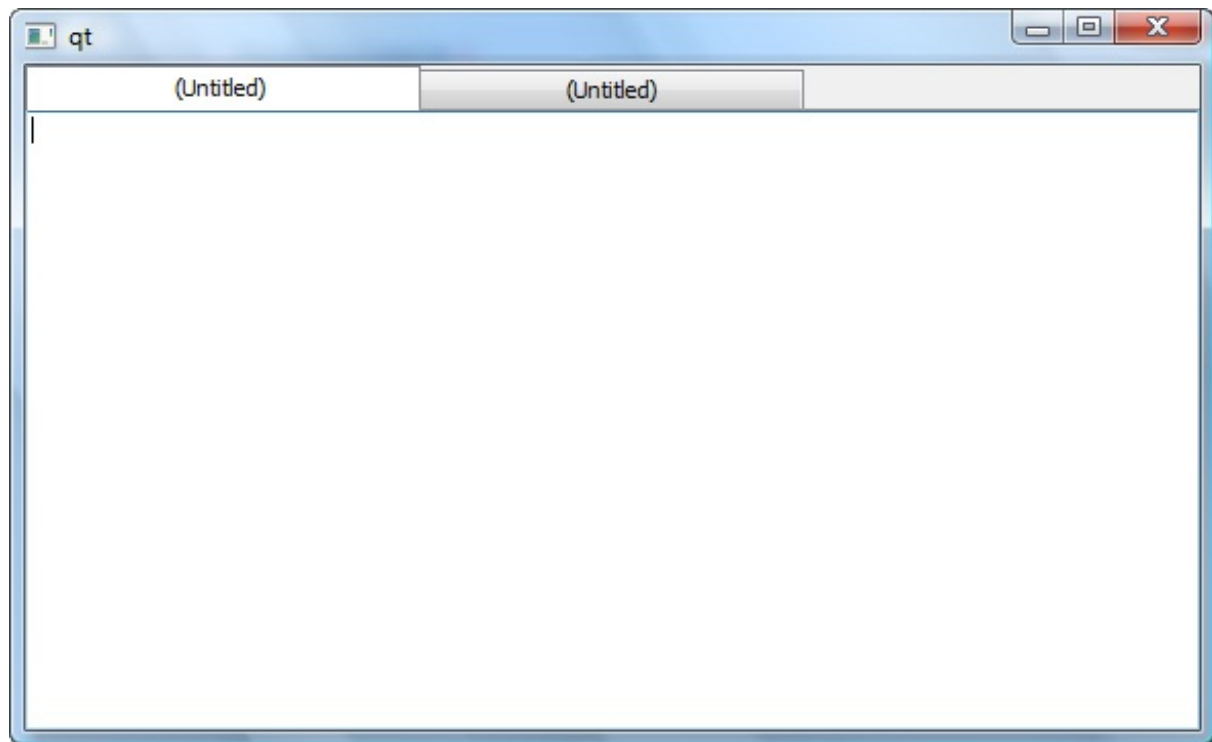
Notons que ce type de fenêtre MDI est de moins en moins utilisée. On a plutôt tendance aujourd'hui à recourir aux onglets lorsque plusieurs documents sont ouverts.

Ça tombe bien, les `QMdiArea` de Qt peuvent changer de mode d'affichage en une seule ligne grâce à `setViewMode()`. Par exemple, avec ce code :

Code : C++

```
zoneCentrale->setViewMode(QMdiArea::TabbedView);
```

... les sous-fenêtres seront organisées sous forme d'onglets :



Ça ne revient pas au même qu'un QTabWidget ça ? 🤔

Si, en effet, on peut faire pareil avec un QTabWidget. L'avantage là c'est que Qt gère chaque onglet comme une fenêtre, et vous pouvez proposer à l'utilisateur de passer en un clic du mode de vue MDI classique au mode onglets. 😊

Les menus

La QMainWindow peut afficher une barre de menus, comme par exemple : Fichier, Edition, Affichage, Aide...
Comment fait-on pour les créer ?

Créer un menu pour la fenêtre principale

La barre de menus est accessible depuis la méthode `menuBar()`. Cette méthode renvoie un pointeur sur un `QMenuBar`, qui vous propose une méthode `addMenu()`. Cette méthode renvoie un pointeur sur le `QMenu` créé.

Puisqu'un petit code vaut tous les discours du monde, voici comment faire :

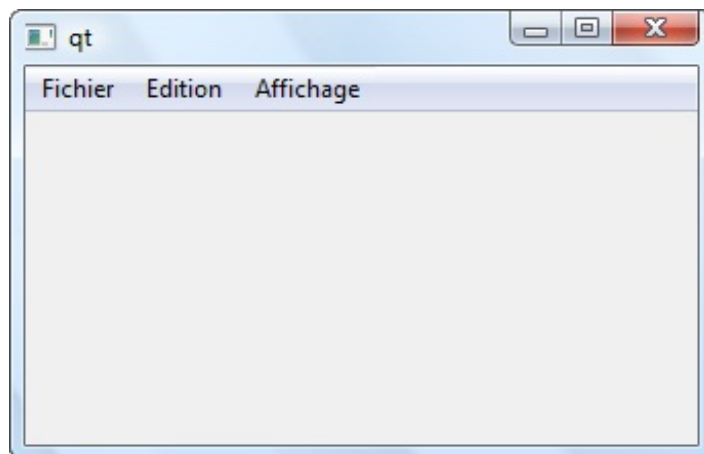
Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");
}
```

Avec ça, nous avons créé 3 menus dont nous gardons les pointeurs (`menuFichier`, `menuEdition`, `menuAffichage`). Vous noterez qu'on utilise ici aussi le symbole `&` pour définir des raccourcis clavier (les lettres F, E et A seront donc des raccourcis vers leurs menus respectifs).

Nous avons maintenant 3 menus sur notre fenêtre :



Mais... ces menus n'affichent rien ! En effet, ils ne contiennent pour le moment aucun élément.

Création d'actions pour les menus

Un **élément de menu** est représenté par une **action**. C'est la classe `QAction` qui gère ça.



Pourquoi avoir créé une classe `QAction` au lieu de... je sais pas moi... `QSubMenu` pour dire "sous-menu" ?

En fait, les `QAction` sont des éléments de menu génériques. Ils peuvent être utilisés à la fois pour les menus et pour la barre d'outils.

Par exemple, imaginons l'élément "Nouveau" qui permet de créer un nouveau document. On peut en général y accéder depuis 2 endroits différents :

- Le menu Fichier / Nouveau.
- Le bouton de la barre d'outils "Nouveau", généralement représenté par une icône de document vide.

Une seule `QAction` peut servir à définir ces 2 éléments à la fois.

Les développeurs de Qt se sont en effet rendus compte que les actions des menus étaient souvent dupliquées dans la barre d'outils, d'où la création de la classe `QAction` que nous réutiliserons lorsque nous créerons la barre d'outils.

Pour créer une action vous avez 2 possibilités :

- Soit vous la créez d'abord, puis vous créez l'élément de menu qui correspond.
- Soit vous créez l'élément de menu directement, et celui-ci vous renvoie un pointeur vers la `QAction` créée automatiquement.

Nous allons tester ces 2 possibilités.

Créer une `QAction`, puis créer l'élément de menu

Nous allons tout d'abord créer une `QAction`, puis nous l'ajouterons au menu "Fichier" :

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
```

```
QMenu *menuFichier = menuBar()->addMenu("&Fichier");

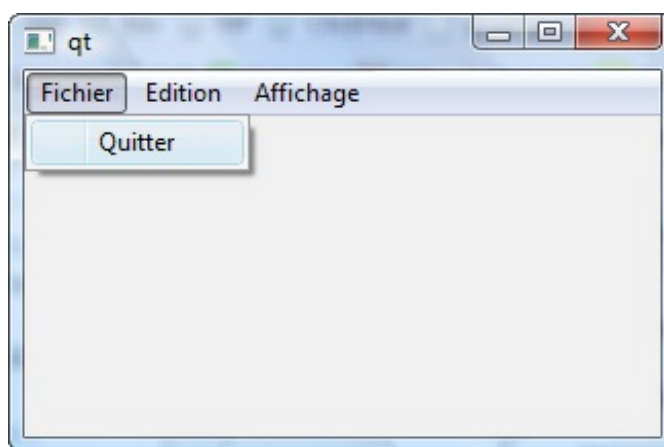
QAction *actionQuitter = new QAction("&Quitter", this);
menuFichier->addAction(actionQuitter);

QMenu *menuEdition = menuBar()->addMenu("&Edition");
QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

}
```

Dans l'exemple de code ci-dessus, nous créons d'abord une QAction correspondant à l'action "Quitter". Nous définissons en second paramètre de son constructeur un pointeur sur la fenêtre principale (this), qui servira de parent à l'action. Puis, nous ajoutons l'action au menu "Fichier".

Résultat, l'élément de menu est créé :



Créer l'élément de menu et récupérer la QAction

Il y a une autre façon de faire.

Parfois, vous trouverez que créer une QAction avant de générer l'élément de menu est un peu lourd. Dans ce cas, vous pouvez passer par une des versions surchargées de la méthode addAction :

Code : C++

```
#include "FenPrincipale.h"

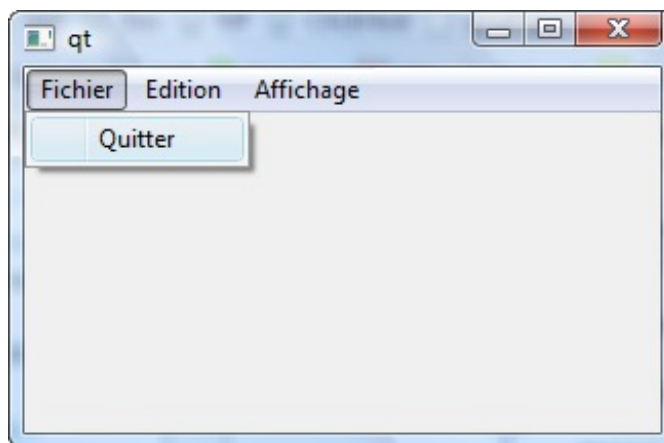
FenPrincipale::FenPrincipale()
{
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");

    QAction *actionQuitter = menuFichier->addAction("&Quitter");

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

Le résultat est strictement le même :



Les sous-menus

Les sous-menus sont gérés par la classe `QMenu`.

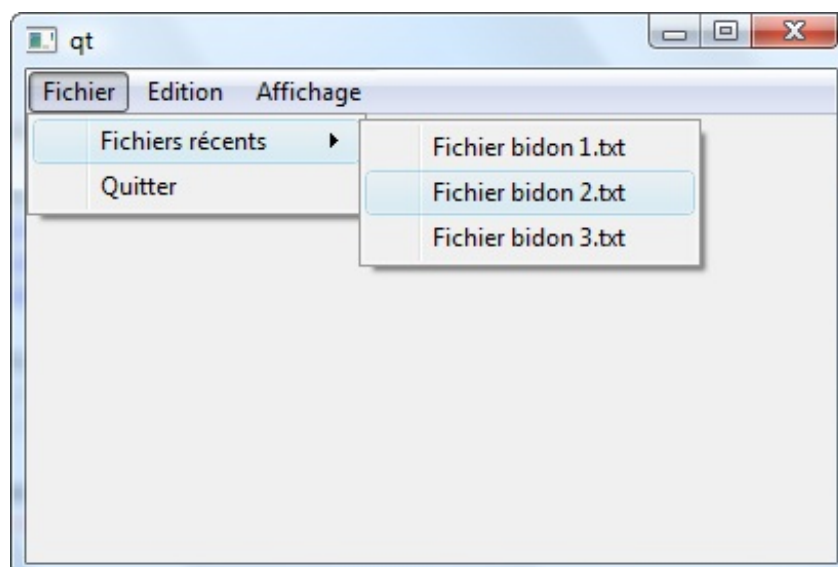
Imaginons que nous voulions créer un sous-menu "Fichiers récents" au menu "Fichier". Ce sous-menu affichera une liste de fichiers récemment ouverts par le programme (des fichiers bidons pour cet exemple).

Au lieu d'appeler `addAction()` de la `QMenuBar`, appelez cette fois `addMenu()` qui renvoie un pointeur vers un `QMenu` :

Code : C++

```
QMenu *fichiersRecents = menuFichier->addMenu("Fichiers &récents");
fichiersRecents->addAction("Fichier bidon 1.txt");
fichiersRecents->addAction("Fichier bidon 2.txt");
fichiersRecents->addAction("Fichier bidon 3.txt");
```

Vous voyez que j'ajoute ensuite de nouvelles actions pour peupler le sous-menu "Fichiers récents". Résultat :



Je n'ai pas récupéré de pointeur vers les `QAction` créées à chaque fois. J'aurais dû le faire si je voulais ensuite connecter les signaux des actions à des slots, mais je ne l'ai pas fait ici pour simplifier le code.



Vous pouvez créer des menus contextuels personnalisés de la même façon, avec des `QMenu`. Un menu contextuel est un menu qui s'affiche lorsqu'on fait un clic droit sur un widget. C'est un petit peu plus complexe. Je vous laisse lire la



doc de QWidget à propos des menus contextuels pour savoir comment faire ça si vous en avez besoin.

Manipulations plus avancées des QAction

Une QAction est au minimum constituée d'un texte descriptif. Mais ce serait dommage de la limiter à ça. Voyons un peu ce qu'on peut faire avec les QAction...

Connecter les signaux et les slots

Le premier rôle d'une QAction est de générer des signaux, que l'on aura connectés à des slots. La QAction propose plusieurs signaux intéressants. Le plus utilisé d'entre eux est triggered() qui indique que l'action a été choisie par l'utilisateur.

On peut connecter notre action "Quitter" au slot quit() de l'application :

Code : C++

```
connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
```

Désormais, un clic sur "Fichier / Quitter" fermera l'application. 😊

Vous avez aussi un événement hovered() qui s'active lorsqu'on passe la souris sur l'action. A tester !

Ajouter un raccourci

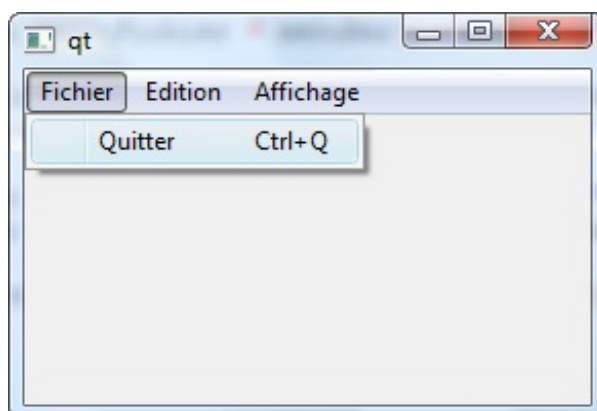
On peut définir un raccourci clavier pour l'action. On passe pour cela par la méthode addShortcut().

Cette méthode peut être utilisée de plusieurs manières différentes. La technique la plus simple est de lui envoyer une [QKeySequence](#) représentant le raccourci clavier :

Code : C++

```
actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
```

Voilà, il suffit d'écrire dans le constructeur de la QKeySequence le raccourci approprié, Qt se chargera de comprendre le raccourci tout seul.



Vous pouvez faire le raccourci clavier Ctrl + Q n'importe où dans la fenêtre maintenant, cela activera l'action "Quitter" !

Sachez que QKeySequence accepte d'autres syntaxes :

Code : C++

```
actionQuitter->setShortcut(QKeySequence(Qt::CTRL + Qt::Key_Q));
```

... créera le même raccourci "Ctrl + Q", sauf que cette fois nous sommes passés par des symboles pour le définir.

Vous pouvez aussi utiliser une séquence prédéfinie, qui s'adapte en fonction des habitudes de l'OS.

Par exemple, la séquence prédéfinie QKeySequence::HelpContents est faite pour représenter un raccourci clavier qui amène à l'aide.

Code : C++

```
actionQuitter->setShortcut(QKeySequence(QKeySequence::HelpContents));
```

Sous Windows, ce sera la touche F1, sous Mac OS X, ce sera le raccourci "Ctrl + ?".

Pour avoir la liste des séquences prédéfinies, c'est dans la doc. 🤔

Ajouter une icône

Chaque action peut avoir une icône.

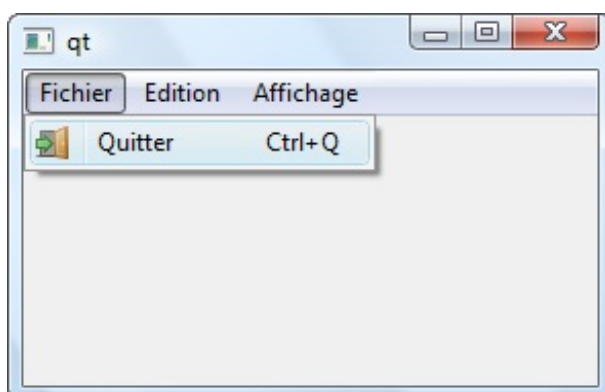
Lorsque l'action est associée à un menu, l'icône est affichée à gauche de l'élément de menu. Mais, souvenez-vous, une action peut aussi être associée à une barre d'outils comme on le verra plus tard. L'icône peut donc aussi être réutilisée dans la barre d'outils.

Pour ajouter une icône, appelez setIcon() et envoyez-lui un QIcon :

Code : C++

```
actionQuitter->setIcon(QIcon("quitter.png"));
```

Résultat :



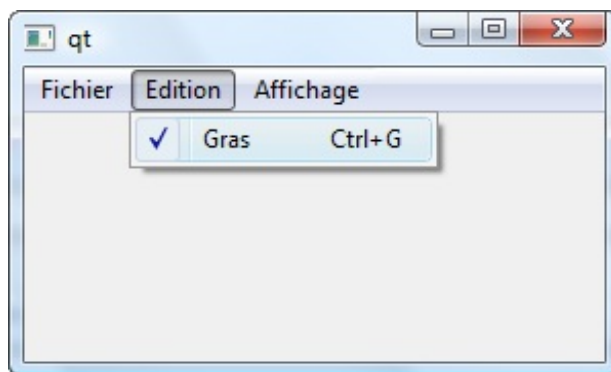
Pouvoir cocher une action

Lorsqu'une action peut avoir 2 états (activée, désactivée), vous pouvez la rendre "cochable" grâce à `setCheckable()`. Imaginons par exemple le menu Edition / Gras :

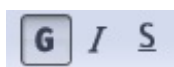
Code : C++

```
actionGras->setCheckable(true);
```

Le menu a maintenant 2 états et peut être précédé d'une case à cocher :



On vérifiera dans le code si l'action est cochée avec `isChecked()`.



Lorsque l'action est utilisée sur une barre d'outils, le bouton reste enfoncé lorsque l'action est "cochée". C'est ce que vous avez l'habitude de voir dans un traitement de texte par exemple (cf image ci-contre 😊).

Ah, puisqu'on parle de barre d'outils, il serait temps d'apprendre à en créer une !

La barre d'outils

La barre d'outils est généralement constituée d'icônes et située sous les menus.

Avec Qt, la barre d'outils utilise des actions pour construire chacun des éléments de la barre. Étant donné que vous avez appris à manipuler des actions juste avant, vous devriez donc être capables de créer une barre d'outils très rapidement. 😊

Pour ajouter une barre d'outils, vous devez tout d'abord appeler la méthode `addToolBar()` de la `QMainWindow`. Il faudra donner un nom à la barre d'outils, même si celui-ci ne s'affiche pas.

Vous récupérez un pointeur vers la `QToolBar` :

Code : C++

```
QToolBar *toolBarFichier = addToolBar("Fichier");
```

Maintenant que nous avons notre `QToolBar`, nous pouvons commencer !

Ajouter une action

Le plus simple est d'ajouter une action à la `QToolBar`. On utilise comme pour les menus une méthode appelée `addAction()` qui prend comme paramètre une `QAction`.

Le gros intérêt que vous devriez saisir maintenant, c'est que vous pouvez réutiliser ici vos `QAction` créées pour les menus !

Code : C++

```
#include "FenPrincipale.h"
```

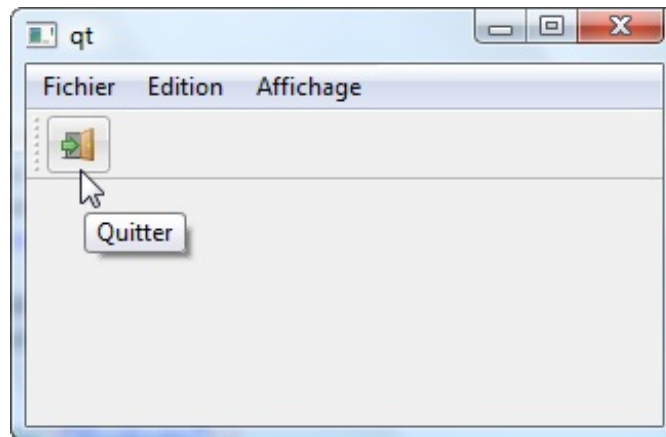
```
FenPrincipale::FenPrincipale()
{
    // Création des menus
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QAction *actionQuitter = menuFichier->addAction("&Quitter");
    actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
    actionQuitter->setIcon(QIcon("quitter.png"));

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    // Création de la barre d'outils
    QToolBar *toolBarFichier = addToolBar("Fichier");
    toolBarFichier->addAction(actionQuitter);

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

Dans ce code, on voit qu'on crée d'abord une QAction pour un menu (ligne 7), puis plus loin on réutilise cette action pour l'ajouter à la barre d'outils (ligne 16).



Comme l'action est la même que celle utilisée pour le menu "Quitter", on retrouve :

- **Son icône** : ici affichée dans la barre d'outils.
- **Son texte** : ici affiché lorsqu'on pointe sur l'icône.
- **Son action** : elle est toujours connectée au slot quit() de l'application, ce qui a pour effet de mettre fin au programme.

Et voilà comment Qt fait d'une pierre deux coups grâce aux QAction ! 😊

Ajouter un widget

Les barres d'outils contiennent le plus souvent des QAction, mais il arrivera que vous ayez besoin d'insérer des éléments plus complexes.

La QToolBar gère justement tous types de widgets.

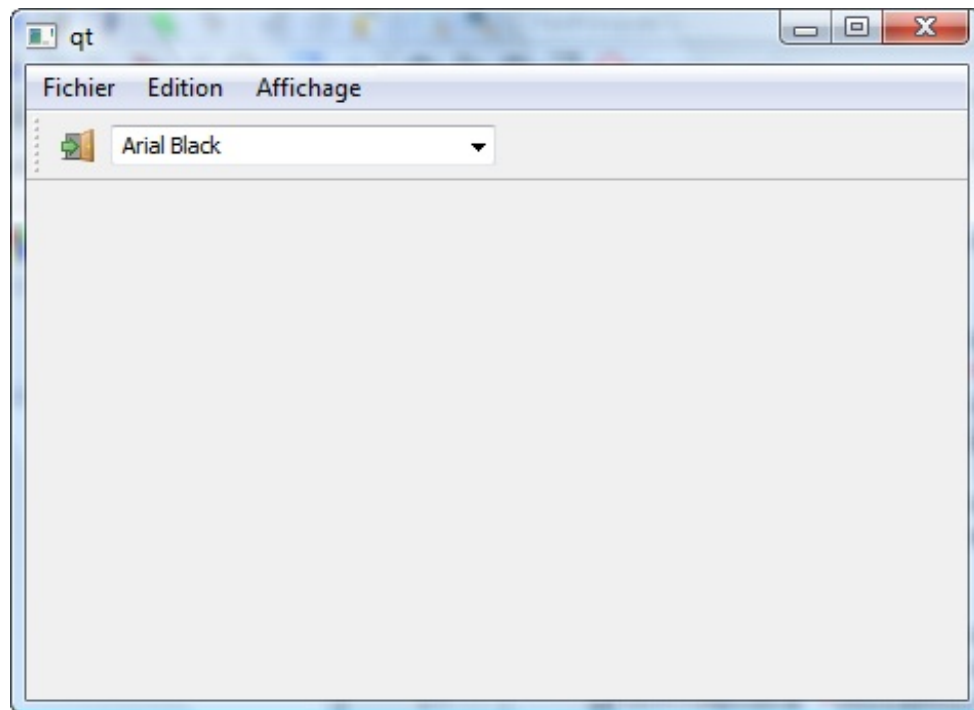
Vous pouvez ajouter des widgets avec la méthode addWidget(), comme vous le faisiez avec les layouts :

Code : C++

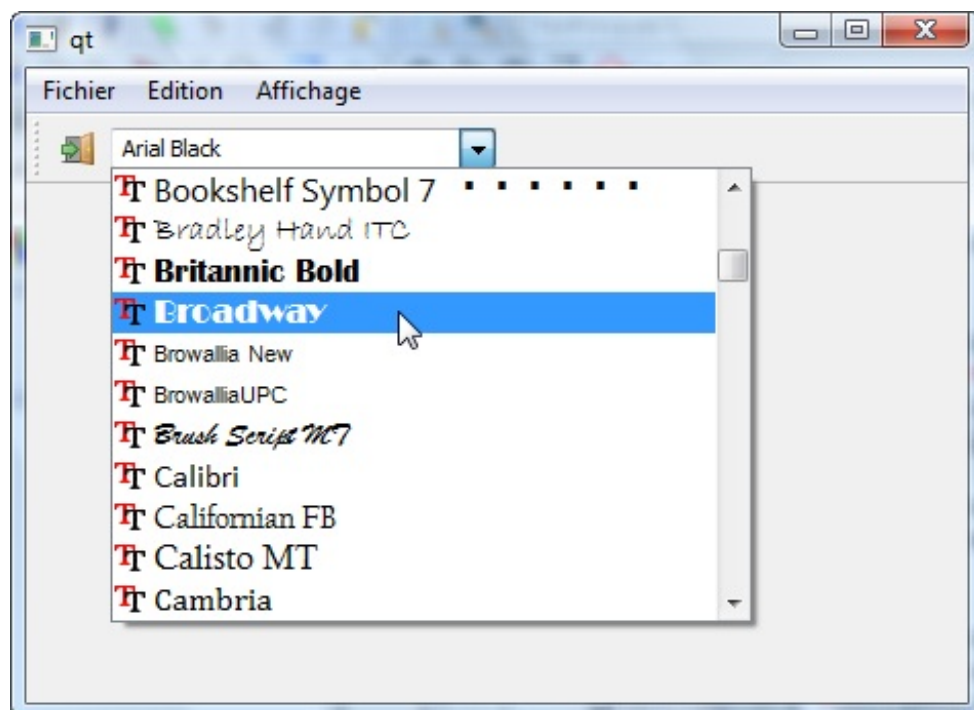
```
QFontComboBox *choixPolice = new QFontComboBox;
toolBarFichier->addWidget(choixPolice);
```

Ici, on insère une liste déroulante (plus précisément une QFontComboBox, une liste déroulante spécialisée dans le choix d'une police).

Le widget s'insère alors dans la barre d'outils :



... et vous pouvez l'utiliser comme n'importe quel widget normal, gérer ses signaux, ses slots, etc.



La méthode `addWidget()` crée une `QAction` automatiquement. Elle renvoie un pointeur vers cette `QAction` créée. Ici, on n'a pas récupéré le pointeur, mais vous pouvez le faire si vous avez besoin d'effectuer des opérations ensuite sur la `QAction`.

Ajouter un séparateur

Si votre barre d'outils commence à comporter trop d'éléments, ça peut être une bonne idée de les séparer. C'est pour cela que Qt propose des *separators* (séparateurs).

Cela vous permettra par exemple de regrouper les boutons "Annuler" et "Rétablir", pour ne pas les confondre avec les boutons "Gras", "Italique", "Souligné".

Il suffit simplement d'appeler la méthode `addSeparator()` au moment où vous voulez insérer un séparateur :

Code : C++

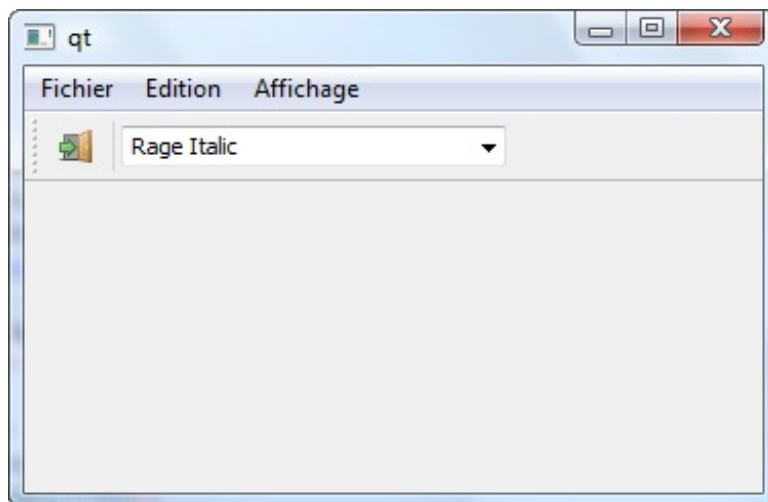
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Création des menus
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QAction *actionQuitter = menuFichier->addAction("&Quitter");
    actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
    actionQuitter->setIcon(QIcon("quitter.png"));

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    // Création de la barre d'outils
    QToolBar *toolBarFichier = addToolBar("Fichier");
    toolBarFichier->addAction(actionQuitter);
    toolBarFichier->addSeparator();
    QFontComboBox *choixPolice = new QFontComboBox;
    toolBarFichier->addWidget(choixPolice);

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

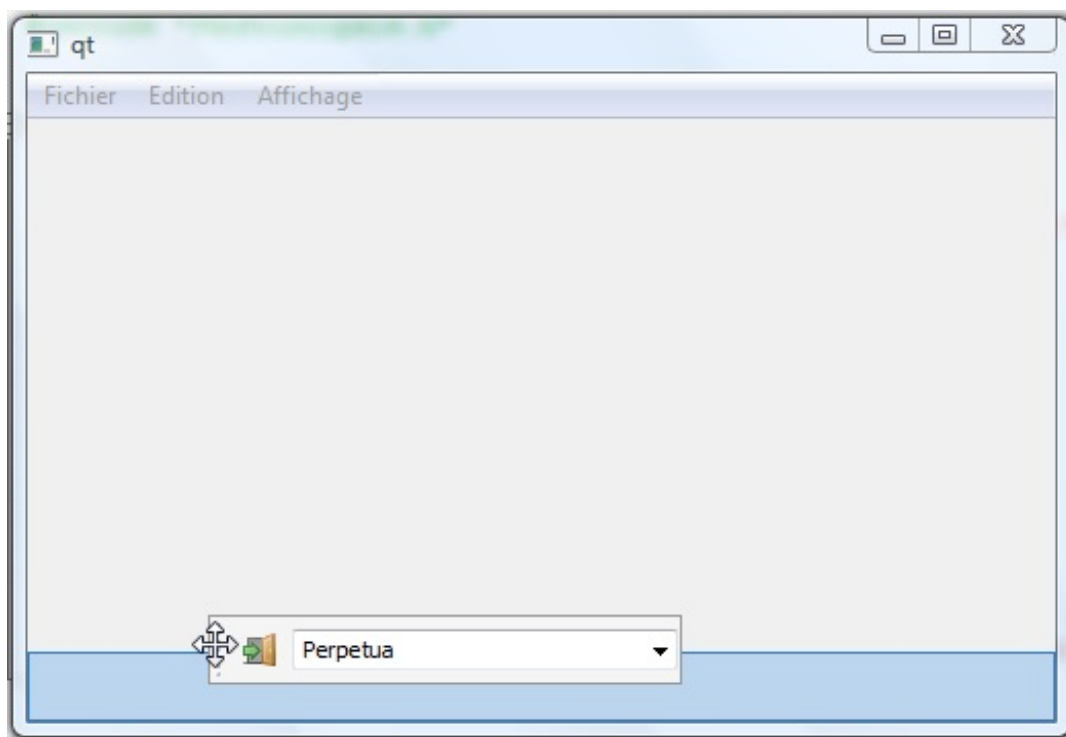


Notez le séparateur entre le bouton Quitter et la liste déroulante

Vous pouvez aussi insérer un séparateur à une position précise grâce à `insertSeparator()`.

Plus d'options pour la barre d'outils

Par défaut, la barre d'outils est déplaçable. Vous pouvez la placer en haut, sur les côtés ou en bas, et vous pouvez même en faire une mini-fenêtre indépendante :



Si vous souhaitez éviter que la barre d'outils soit déplaçable, modifiez sa propriété `movable`.

Si vous souhaitez juste éviter qu'on puisse créer une mini-fenêtre indépendante, modifiez sa propriété `floatable`.

Les docks

Les docks sont des mini-fenêtres que l'on peut généralement déplacer à notre guise dans la fenêtre principale.

Vous avez l'habitude d'en voir dans des programmes complexes comme Photoshop ou votre IDE (Code::Blocks en utilise un pour afficher la liste des fichiers du projet par exemple).

Créer un QDockWidget

Créez un `QDockWidget` et affectez-lui un titre pour commencer. Indiquez que la fenêtre principale (`this`) est son widget parent en second paramètre :

Code : C++

```
QDockWidget *dock = new QDockWidget("Palette", this);
```

Puis, placez ce dock sur la fenêtre principale à l'aide de la méthode `addDockWidget()` :

Code : C++

```
addDockWidget(Qt::LeftDockWidgetArea, dock);
```

Comme vous pouvez le voir, le premier paramètre permet d'indiquer à quel endroit le dock doit être placé. On peut le mettre en haut, en bas, à gauche ou à droite. On peut même faire flotter le dock comme une mini-fenêtre.

Par défaut, l'utilisateur a le droit de déplacer le dock. Si vous ne souhaitez pas qu'il puisse le faire, faites appel à la méthode `setFeatures()` du `QDockWidget`. Vous pouvez tout personnaliser à partir de là.



De manière générale, évitez d'empêcher à l'utilisateur de réorganiser les docks. La puissance des docks c'est justement que l'on peut les réorganiser selon ses préférences. Ce serait dommage de perdre en fonctionnalités.

Peupler le QDockWidget

Notre QDockWidget est créé mais il ne contient rien.

Pour le peupler en widgets, il faut d'abord créer un widget conteneur et indiquer que ce widget gère le contenu du QDockWidget.

Code : C++

```
QWidget *contenuDock = new QWidget;
dock->setWidget(contenuDock);
```

Après, il n'y a plus qu'à placer des widgets et / ou des layouts dans contenuDock, comme si c'était une fenêtre. Je mets quelques widgets au pif, ne faites pas spécialement attention au détail de ce code :

Code : C++

```
QPushButton *crayon = new QPushButton("Crayon");
QPushButton *pinceau = new QPushButton("Pinceau");
QPushButton *feutre = new QPushButton("Feutre");
QLabel *labelEpaisseur = new QLabel("Epaisseur :");
QSpinBox *epaisseur = new QSpinBox;

QVBoxLayout *dockLayout = new QVBoxLayout;
dockLayout->addWidget(crayon);
dockLayout->addWidget(pinceau);
dockLayout->addWidget(feutre);
dockLayout->addWidget(labelEpaisseur);
dockLayout->addWidget(epaisseur);

contenuDock->setLayout(dockLayout);
```

Code complet

Voici un code complet qui utilise des menus, une barre d'outils et un dock. Il est un peu long mais ne vous laissez pas impressionner, il n'y a rien de nouveau ni de compliqué.

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Création des menus
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");

    QAction *actionQuitter = menuFichier->addAction("&Quitter");
    actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
    actionQuitter->setIcon(QIcon("quitter.png"));

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    // Création de la barre d'outils
    QToolBar *toolBarFichier = addToolBar("Fichier");
    toolBarFichier->addAction(actionQuitter);
    toolBarFichier->addSeparator();
    QFontComboBox *choixPolice = new QFontComboBox;
```

```

        toolBarFichier->addWidget(choixPolice);

        // Création des docks
        QDockWidget *dock = new QDockWidget("Palette", this);
        addDockWidget(Qt::LeftDockWidgetArea, dock);

        QWidget *contenuDock = new QWidget;
        dock->setWidget(contenuDock);

        QPushButton *crayon = new QPushButton("Crayon");
        QPushButton *pinceau = new QPushButton("Pinceau");
        QPushButton *feutre = new QPushButton("Feutre");
        QLabel *labelEpaisseur = new QLabel("Epaisseur :");
        QSpinBox *epaisseur = new QSpinBox;

        QVBoxLayout *dockLayout = new QVBoxLayout;
        dockLayout->addWidget(crayon);
        dockLayout->addWidget(pinceau);
        dockLayout->addWidget(feutre);
        dockLayout->addWidget(labelEpaisseur);
        dockLayout->addWidget(epaisseur);

        contenuDock->setLayout(dockLayout);

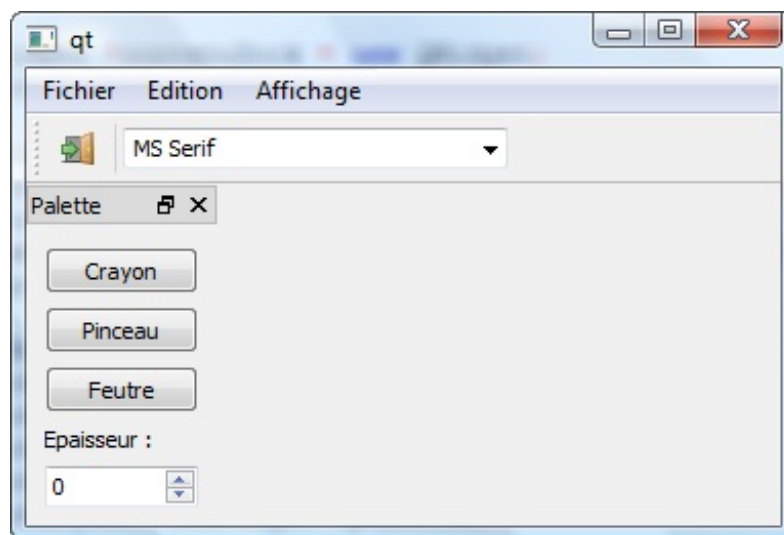
        // Création de la zone centrale
        QWidget *zoneCentrale = new QWidget;
        setCentralWidget(zoneCentrale);

        connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
    }

```

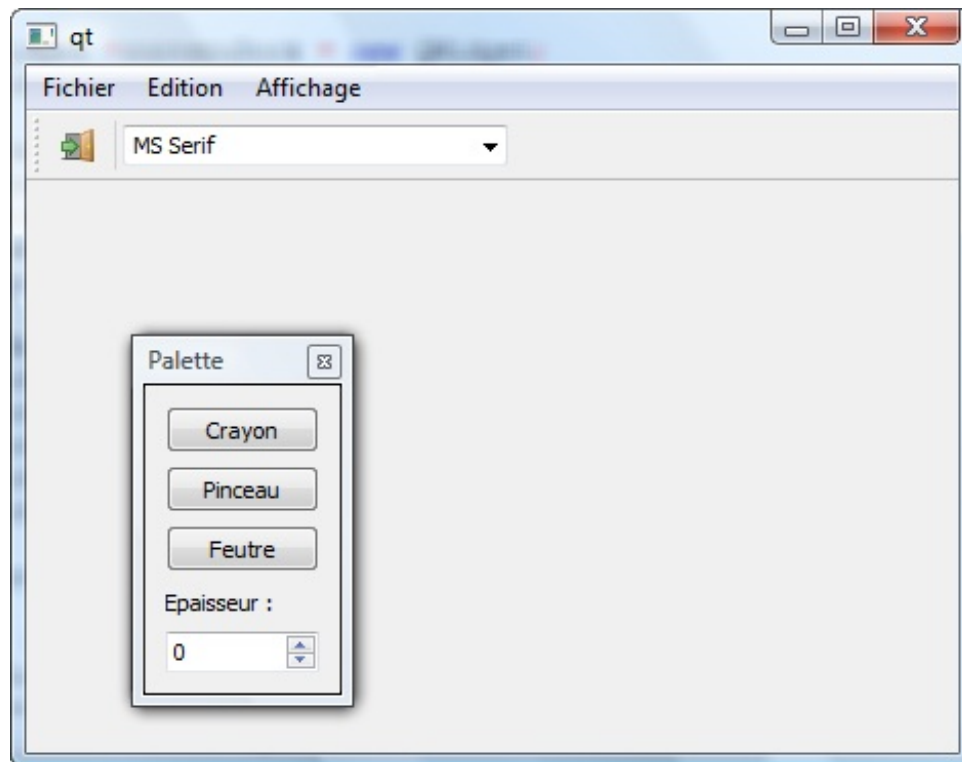
J'ai volontairement mis des commentaires pour séparer les différentes sections de la génération de la fenêtre.

Voilà le résultat :



Le dock est placé à gauche comme nous l'avons demandé, mais nous pouvons le changer de côté.

D'autre part, nous pouvons faire sortir le dock et nous en servir comme d'une mini-fenêtre indépendante :



Bien sûr, cet exemple est à améliorer : il vaudrait mieux afficher des icônes sur les boutons plutôt que du texte, c'est plus intuitif et plus habituel. 😊

La barre d'état

La barre d'état est une petite barre affichée en bas de la fenêtre. Elle indique ce qu'est en train de faire l'application.

Par exemple, un navigateur web comme Firefox affiche le message "Terminé" lorsque la page web a été chargée. Lorsque la page est en cours de chargement, une barre de progression apparaît à cet emplacement.

La barre d'état est automatiquement créée et retournée par la méthode `statusBar()` de la `QMainWindow`. Celle-ci renvoie un pointeur vers une `QStatusBar` que vous devez conserver :

Code : C++

```
QStatusBar *barreEtat = statusBar();
```

Notre barre d'état est créée ! 😊

Maintenant, que peut-on faire dedans ?

Il faut savoir qu'une barre d'état peut afficher 3 types de messages différents :

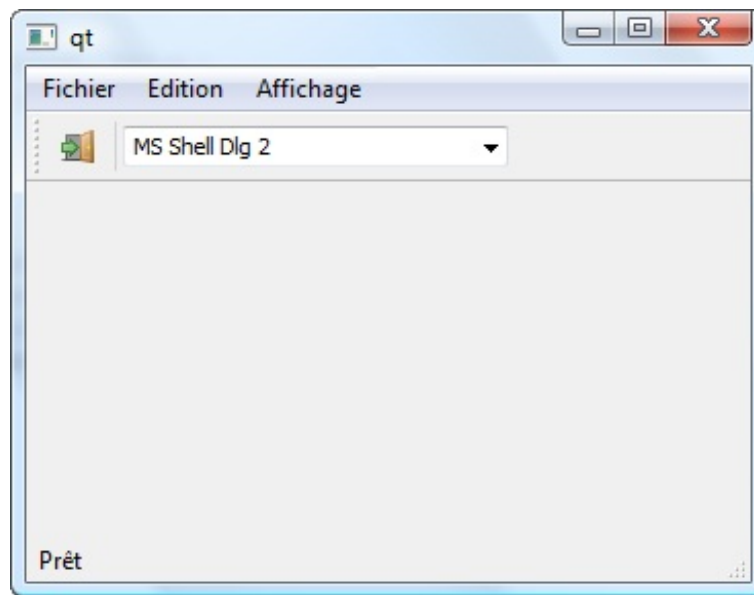
- **Un message temporaire** : il est affiché brièvement par-dessus tous les messages normaux.
- **Un message normal** : il est affiché tout le temps, sauf quand un message temporaire est affiché.
- **Un message permanent** : il est affiché tout le temps, même quand un message temporaire est affiché.

Les messages temporaires

C'est le plus simple : appelez la méthode `showMessage()` et indiquez le message à afficher. Par exemple :

Code : C++

```
barreEtat->showMessage ("Prêt");
```



Vous pouvez indiquer en second paramètre la durée d'affichage du message en millisecondes, avant qu'il disparaisse. Exemple :

Code : C++

```
barreEtat->showMessage ("Le fichier a été sauvegardé", 2000);
```

Ce message restera affiché 2 secondes.

Les messages normaux et permanents

Pour les messages normaux et permanents, c'est un peu plus compliqué : il faut utiliser des widgets. Vous pouvez placer a priori n'importe quel widget dans cette zone, les plus courants étant les QLabel et les QProgressBar.

Utilisez :

- **addWidget()** : pour afficher un widget normal.
- **addPermanentWidget()** : pour afficher un widget permanent.

Exemple de widget normal affiché en barre d'état :

Code : C++

```
QProgressBar *progression = new QProgressBar;  
barreEtat->addWidget (progression);
```

Voilà, avec ça vous pouvez faire tout ce que vous voulez. 😊

Les status tips des QAction

Les QAction disposent d'une propriété dont je ne vous ai pas parlé jusqu'ici : statusTip.

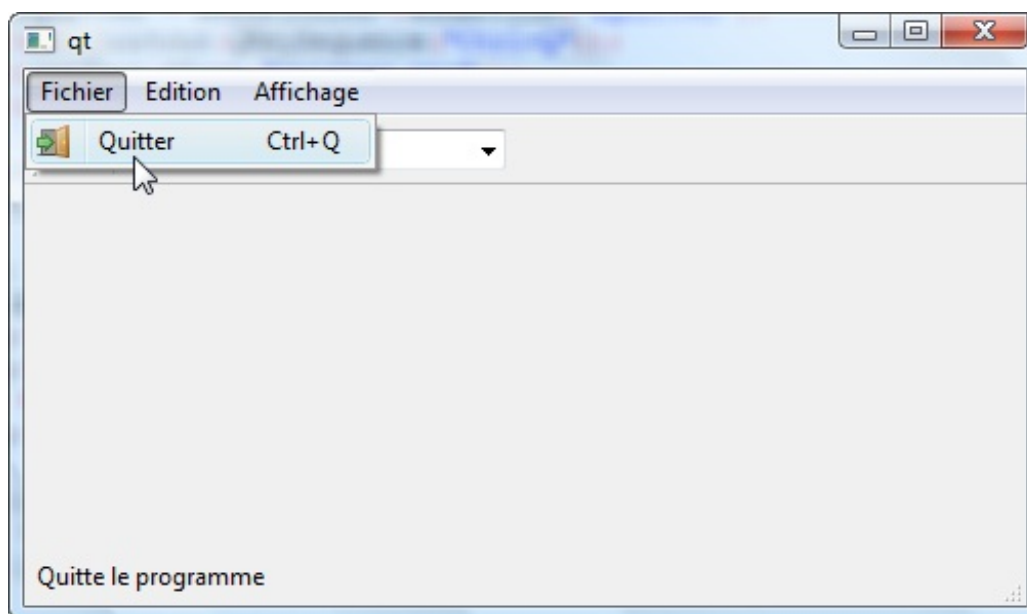
Elle permet d'indiquer un message qui doit s'afficher dans la barre d'état lorsqu'on pointe sur une action. Cela permet de donner de plus amples indications sur le rôle d'un élément de menu par exemple.

Mettons en place un statusTip sur l'action "Quitter" :

Code : C++

```
actionQuitter->setStatusTip("Quitte le programme");
```

Et voici ce que fait le statusTip, lorsqu'on pointe sur l'élément de menu :



Vous pouvez voir que la barre d'état affiche le statusTip lorsque la souris pointe sur l'action. 😊

Pfiou ! Ce n'est pas la fenêtre **principale** pour rien : c'est fou tout ce qu'on peut mettre dedans. A tel point qu'on se demande des fois s'il reste de la place pour afficher le widget central. 🤔

La QMainWindow offre beaucoup de possibilités, mais ne vous sentez pas obligés de toutes les utiliser. Au contraire, ne faites appel qu'à ce qui vous sert, et ne surchargez pas inutilement la fenêtre principale. Sinon, vos utilisateurs mettront du temps avant d'arriver à la maîtriser, ce qui n'est jamais très bon.

Exercice : créer un éditeur de texte

Je crois que l'exercice de ce chapitre est tout trouvé : je vous propose de faire un éditeur de texte (en mode MDI, *of course* 😊). En effet, un éditeur de texte contient un peu tout ce qu'on vient d'apprendre :

- Des menus
- Une barre d'outils
- Une barre d'état
- Un mode multi-documents (MDI)

Et vous pouvez même ajouter des docks si vous leur trouvez une utilité. 😊

Bien que classique, c'est un très bon exercice. Il faudra bien vous organiser. Pour ce qui est de l'écriture des fichiers, utilisez [QFile](#).

Bon courage !

Traduire son programme avec Qt Linguist

Si vous avez de l'ambition pour vos programmes, vous aurez peut-être envie un jour de les traduire dans d'autres langues. En effet, ce serait dommage de limiter votre programme seulement aux francophones, il y a certainement de nombreuses autres personnes qui aimeraient pouvoir en profiter !

La traduction d'applications n'est normalement pas une chose facile. D'ailleurs, il ne s'agit pas seulement de traduire des mots ou des phrases. Il ne suffit pas toujours de dire "Ouvrir" = "Open".

En effet, on parle des milliers de langues et dialectes différents sur notre bonne vieille planète. Certaines ressemblent au français, certaines écrivent de droite à gauche (l'arabe par exemple), d'autres ont des accents très particuliers que nous n'avons pas l'habitude d'utiliser (le ñ espagnol par exemple). Et je vous parle même pas des caractères hébraïques et japonais. 🤔

On ne parle donc pas seulement de traduction mais de **localisation**. Il faut que notre programme puisse s'adapter avec les habitudes de chaque langue. Il faut que les phrases soient écrites de droite à gauche si nécessaire.

C'est là que Qt excelle et vous simplifie littéralement la tâche. Tout est prévu. Tous les outils pour traduire au mieux vos applications sont installés de base.

Comment ça fonctionne ? Nous allons voir ça. 😊

Les étapes de la traduction

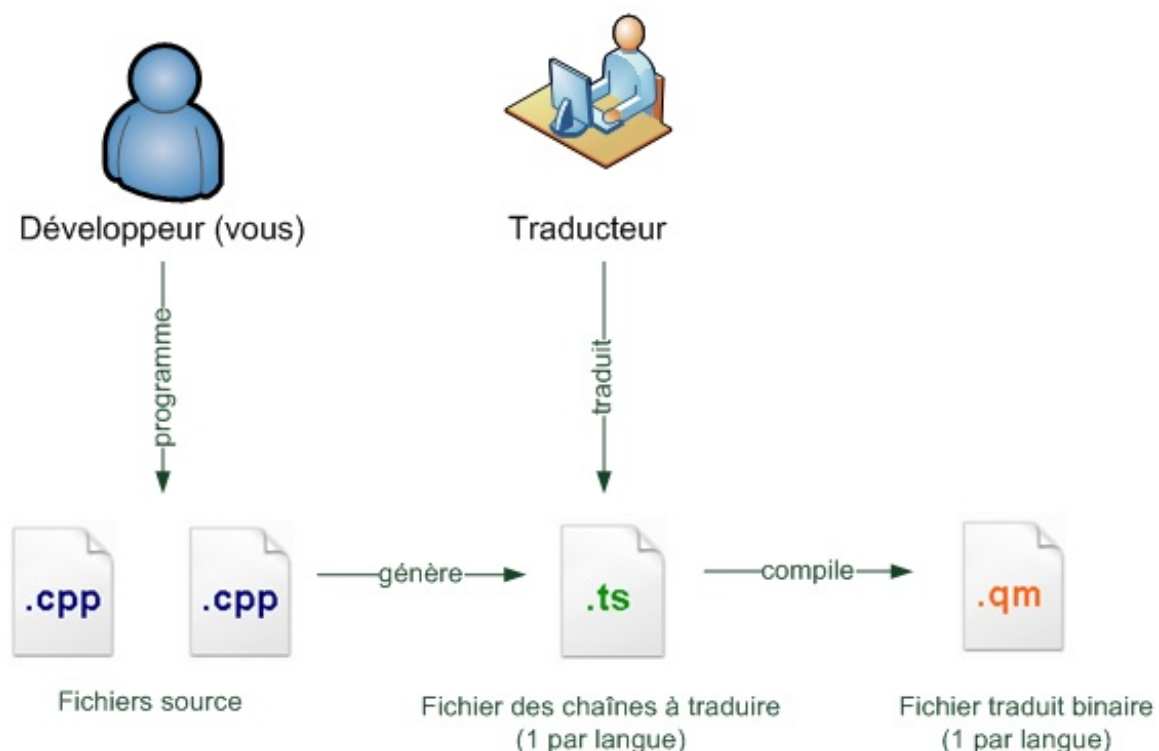
La traduction de programmes Qt est un processus bien pensé... mais encore faut-il comprendre comment ça fonctionne. 🤔

Qt suppose que les développeurs (vous) ne sont pas des traducteurs. Il suppose donc que ce sont 2 personnes différentes. Tout a été fait pour que les traducteurs, même si ce ne sont pas des informaticiens, soient capables de traduire votre programme.



Dans la pratique, si c'est un petit projet personnel, vous serez peut-être *aussi* le traducteur de votre programme. Mais nous allons supposer ici que le traducteur est une autre personne.

Je vous propose de regarder ce schéma de mon crû qui résume grosso modo les étapes de la traduction :



1. Tout d'abord, il y a le développeur. C'est vous. Vous écrivez normalement votre programme, en rédigeant les messages dans le code source dans votre langue maternelle (le français). Bref, rien ne change, à part un ou deux petits détails dont on reparlera dans la prochaine sous-partie.
2. Puis, vous générez un fichier contenant les chaînes à traduire. Un programme livré avec Qt le fait automatiquement pour

vous. Ce fichier porte l'extension .ts, et est généralement de la forme : nomduprogramme_langue.ts.

Par exemple, pour le ZeroClassGenerator, ça donne quelque chose comme zeroclassgenerator_en.ts pour la traduction anglaise, zeroclassgenerator_es.ts pour la traduction espagnole, etc. Il faut connaître le symbole à 2 lettres de la langue de destination pour donner un nom correct au fichier .ts. Généralement c'est la même chose que les extensions des noms de domaine : fr (français), pl (polonais), ru (russe)...

3. Le traducteur récupère le ou les fichiers .ts à traduire (un par langue). Il les traduit via le programme **Qt Linguist** qu'on découvrira dans quelques minutes.
4. Une fois que le traducteur a fini, il retourne les fichiers .ts traduits au développeur, qui les "compile" en fichiers .qm binaires. La différence entre un .ts et un .qm, c'est un peu comme la différence entre un .cpp (la source) et un .exe (le programme binaire final).
Le .qm contenant les traductions au format binaire, Qt pourra le charger et le lire très rapidement lors de l'exécution du programme, ce qui fait qu'on ne sentira pas de ralentissement si on charge une version traduite du programme.

Je vous propose de découvrir pas à pas chacune de ces étapes dans ce chapitre. 😊

Nous allons commencer par vous, le développeur. Que faut-il faire de spécial lorsque vous écrivez le code source du programme ?

Préparer son code à une traduction

La toute première étape de la traduction consiste à écrire son code de manière adaptée, afin que des traducteurs puissent ensuite récupérer tous les messages à traduire.

Utilisez QString pour manipuler des chaînes de caractères

Comme vous le savez déjà, Qt utilise exclusivement sa classe **QString** pour gérer les chaînes de caractères. Cette classe, très complète, gère nativement l'Unicode.



L'Unicode est une norme qui indique comment sont gérés les caractères à l'intérieur de l'ordinateur. Elle permet à un ordinateur d'afficher sans problème tous types de caractères, en particulier les caractères étrangers.

[En savoir plus sur Unicode.](#)

QString n'a donc aucun problème pour gérer des alphabets cyrilliques ou arabes.

C'est pourquoi il est recommandé, si votre application est susceptible d'être traduite, d'utiliser toujours des QString pour manipuler des chaînes de caractères.

Code : C++

```
QString chaîne = "Bonjour"; // Bon : adapté pour la traduction
char chaîne[] = "Bonjour"; // Mauvais : inadapté pour la traduction
```

Voilà, c'est juste un conseil que je vous donne là : de manière générale, utilisez autant que possible des QString. Evitez les tableaux de char.

Faites passer les chaînes à traduire par la méthode tr()

Utilisation basique

La méthode tr() permet d'indiquer qu'une chaîne devra être traduite. Par exemple, avant vous faisiez :

Code : C++

```
quitter = new QPushButton("&Quitter");
```


Cela ne permettra pas de traduire le texte du bouton. En revanche, si vous faites d'abord appel à la méthode `tr()` :

Code : C++

```
quitter = new QPushButton(tr("&Quitter"));
```

... alors le texte pourra être traduit ensuite. 😊

Vous rédigez donc les textes de votre programme dans votre langue maternelle (ici le français) en les entourant d'un `tr()`.



La méthode `tr()` est définie dans `QObject`. C'est donc une méthode statique dont héritent toutes les classes de Qt, puisqu'elles dérivent de `QObject`. Dans la plupart des cas, écrire `tr()` devrait donc fonctionner. Si toutefois vous n'êtes pas dans une classe qui hérite de `QObject`, il faudra faire précéder `tr()` de `QObject::`, comme ceci :

```
quitter = new QPushButton(QObject::tr("&Quitter"));
```

Facultatif : ajouter un message de contexte

Parfois, il arrivera que le texte à traduire ne soit pas suffisamment explicite à lui tout seul, ce qui rendra difficile sa traduction pour le traducteur qui ne le verra pas dans son contexte.

Vous pouvez ajouter en second paramètre un message pour expliquer le contexte au traducteur.

Code : C++

```
quitter = new QPushButton(tr("&Quitter", "Utilisé pour le bouton de fermeture"));
```

Ce message ne sera pas affiché dans votre programme : il aidera juste le traducteur à comprendre ce qu'il doit traduire. En effet, dans certaines langues "Quitter" se traduit peut-être de plusieurs manières différentes. Avec le message de contexte, le traducteur saura comment bien traduire le mot.

En général, le message de contexte n'est pas obligatoire.

Parfois cependant, il devient vraiment indispensable. Par exemple quand on doit traduire un raccourci clavier (eh oui !) :

Code : C++

```
actionQuitter->setShortcut(QKeySequence(tr("Ctrl+Q", "Raccourci clavier pour quitter")));
```

Le traducteur pourra ainsi traduire la chaîne en "Ctrl+S" si c'est le raccourci adapté dans la langue de destination.

Facultatif : gestion des pluriels

Parfois, une chaîne doit être écrite différemment selon le nombre d'éléments.

Imaginons un programme qui lit le nombre de fichiers dans un répertoire. Il affiche le message "*Il y a X fichier(s)*". Comment traduire ça correctement ?

En fait, ça dépend vraiment des langues. Le pluriel est géré différemment en anglais et en français par exemple :

Nombre	Français	Anglais
--------	----------	---------

0	Il y a 0 fichier.	There are 0 files.
1	Il y a 1 fichier.	There is 1 file.
2	Il y a 2 fichiers.	There are 2 files.

Comme vous pouvez le voir, en français on dit "Il y a 0 fichier.", et en anglais "There are 0 files.". Les anglais mettent du pluriel quand le nombre d'éléments est à 0 !

Et encore, je vous parle pas des russes, qui ont un pluriel pour quand il y a 2 éléments et un autre pluriel pour quand il y en a 3 ! (je simplifie parce qu'en fait c'est même un peu plus compliqué que ça encore)



J'ai jamais m'en sortir avec tous ces cas à gérer ! 😓

Eh bien si, rassurez-vous. Qt est capable de gérer tous les pluriels pour chacune des langues. Ce sera bien entendu le rôle du traducteur ensuite de traduire ces pluriels correctement.

Comment faire ? Utilisez le 3ème paramètre facultatif qui indique la cardinalité (le nombre d'éléments).

Exemple :

Code : C++

```
tr("Il y a %n fichier(s)", "", nombreFichiers);
```



Si on ne veut pas indiquer de contexte comme moi dans ce cas, on est quand même obligé d'envoyer une chaîne vide pour utiliser le 3ème paramètre (c'est la règle des paramètres facultatifs en C++).

Qt utilisera automatiquement la bonne version du texte traduit selon la langue de destination et le nombre d'éléments. Par ailleurs, le %n sera remplacé par le nombre indiqué en 3ème paramètre.

Bon, avec tout ça, votre programme est codé de manière à pouvoir être traduit.

Maintenant, comment se passe l'étape de la traduction ?

Créer les fichiers de traduction .ts

Nous avons maintenant un programme qui fait appel à la méthode tr() pour désigner toutes les chaînes de caractères qui doivent être traduites.

On va prendre l'exemple de notre TP ZeroClassGenerator. Je l'ai adapté pour qu'il utilise des tr().

On souhaite que ZeroClassGenerator soit traduit dans les langues suivantes :

- Anglais
- Espagnol

Nous devons générer 2 fichiers de traduction :

- zeroclassgenerator_en.ts pour l'anglais
- zeroclassgenerator_es.ts pour l'espagnol

Il va falloir éditer le fichier .pro. Celui-ci se trouve dans le dossier de votre projet et a normalement été généré automatiquement par Qt Creator.

Ouvrez ce fichier (dans mon cas ZeroClassGenerator.pro) avec un éditeur de texte comme Bloc-Notes, ou Notepad++, ou ce que

vous voulez.

Pour le moment il devrait contenir quelque chose comme ça :

Code : Autre

```
#####
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenCodeGenere.h FenPrincipale.h
SOURCES += FenCodeGenere.cpp FenPrincipale.cpp main.cpp
```

Rajoutez à la fin une directive TRANSLATIONS en indiquant les noms des fichiers de traduction à générer. Ici, nous rajoutons un fichier pour la traduction anglaise, et un autre pour la traduction espagnole :

Code : Autre

```
#####
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenCodeGenere.h FenPrincipale.h
SOURCES += FenCodeGenere.cpp FenPrincipale.cpp main.cpp
TRANSLATIONS = zeroclassgenerator_en.ts zeroclassgenerator_es.ts
```

Bien. Maintenant, nous allons faire appel à un programme en console de Qt qui permet de générer automatiquement les fichiers .ts.

Ouvrez une console (sous Windows, utilisez le raccourci Qt Command Prompt). Allez dans le dossier de votre projet. Tapez :

Code : Console

```
lupdate NomDuProjet.pro
```

lupdate est un programme qui va mettre à jour les fichiers de traduction .ts, ou les créer si ceux-ci n'existent pas.

Essayons d'exécuter lupdate sur ZeroClassGenerator.pro :

Code : Console

```
C:\Users\Mateo\Projets\ZeroClassGenerator>lupdate ZeroClassGenerator.pro
Updating 'zeroclassgenerator_en.ts'...
Found 17 source text(s) (17 new and 0 already existing)
Updating 'zeroclassgenerator_es.ts'...
```

```
Found 17 source text(s) (17 new and 0 already existing)
```

Le programme lupdate a trouvé dans mon code source 17 chaînes à traduire. Il a vérifié si les .ts existaient (ce qui n'était pas le cas) et les a donc créés.

Ce programme est intelligent puisque, si vous l'exécutez une seconde fois, il ne mettra à jour que les chaînes qui ont changé. C'est très pratique, puisque cela permet d'envoyer au traducteur seulement ce qui a changé par rapport à la version précédente de votre programme !

Vous devriez maintenant avoir 2 fichiers supplémentaires dans le dossier de votre projet : zeroclassgenerator_en.ts et zeroclassgenerator_es.ts.

Envoyez-les à votre traducteur (s'il sait parler anglais et espagnol 🤪). Bien entendu, le fait qu'on ait 2 fichiers distincts nous permet d'envoyer le premier à un traducteur anglais, et le second à un traducteur espagnol.

Traduire l'application sous Qt Linguist

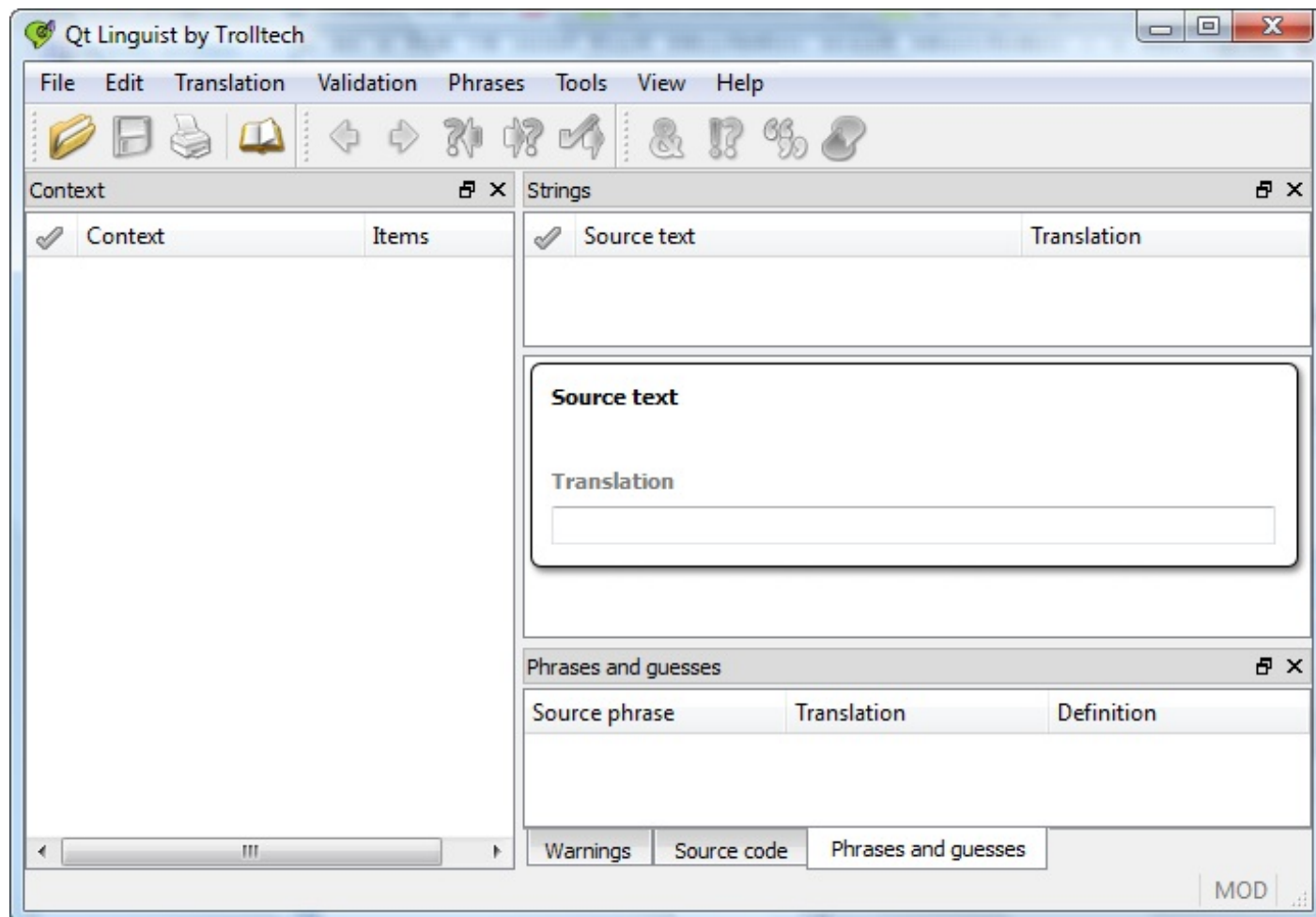


Qt a installé plusieurs programmes, vous vous souvenez ?
L'un d'eux va nous être sacrément utile maintenant : c'est **Qt Linguist**.

Votre traducteur a besoin de 2 choses :

- Du fichier .ts à traduire
- Et de Qt Linguist pour pouvoir le traduire !

Votre traducteur lance donc Qt Linguist. Il devrait voir quelque chose comme ça :



Ca a l'air un petit peu compliqué (et encore, vous avez pas vu Qt Designer 😊).

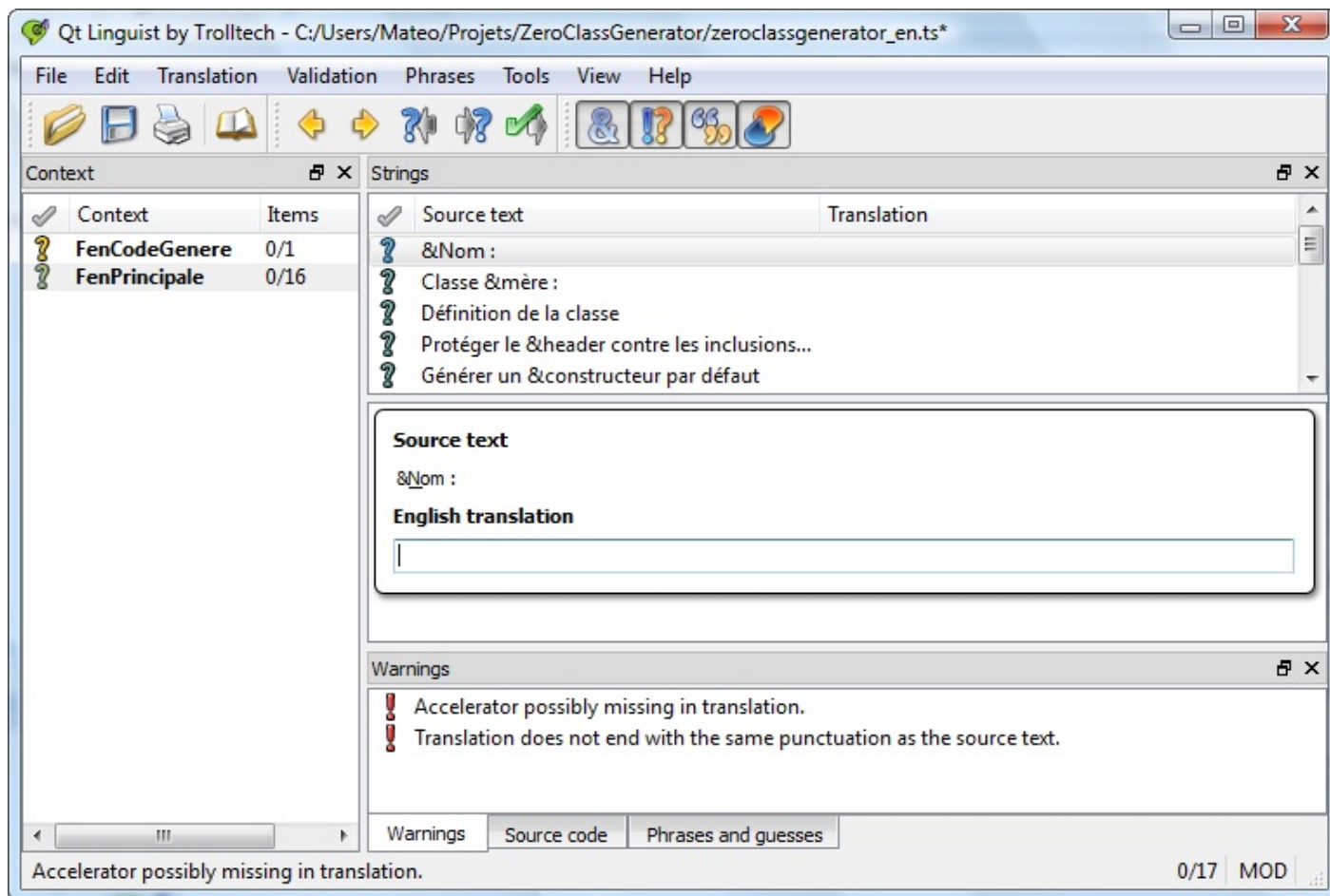
Vous reconnaissez d'ailleurs sûrement une QMainWindow, avec une barre de menus, une barre d'outils, des docks, et un widget central (bah oui, les programmes livrés avec Qt sont faits avec Qt 😊).

En fait, les docks prennent tellement de place qu'on a du mal à savoir où est le widget central. Pour vous aider, c'est l'espèce de bulle ronde au milieu à droite, avec "Source text" et "Translation". C'est justement là qu'on traduira les chaînes de caractères.



Comme vous le savez déjà, les docks peuvent être déplacés. N'hésitez pas à arranger la fenêtre à votre guise. Vous pouvez même faire sortir les docks de la fenêtre principale pour en faire des mini-fenêtres flottantes.

Ouvrez un des fichiers .ts avec Qt Linguist, par exemple zeroclassgenerator_en.ts. La fenêtre se remplit :



Détaillons un peu chaque section de la fenêtre :

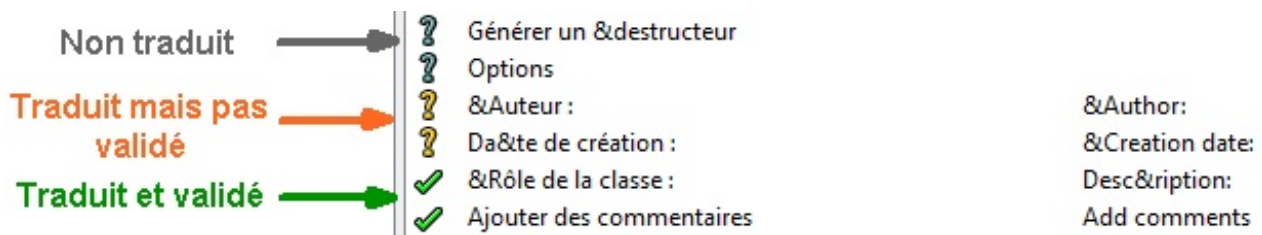
- **Context**: affiche la liste des fichiers source qui contiennent des chaînes à traduire. Vous reconnaissez vos fenêtres FenCodeGenere et FenPrincipale. Le nombre de chaînes traduites est indiqué à droite.
- **Strings** : c'est la liste des chaînes à traduire pour le fichier sélectionné. Ces chaînes ont été extraites grâce à la présence de la méthode tr().
- **Au milieu** : vous avez la version française de la chaîne, et on vous demande d'écrire la version anglaise. Notez que Qt a automatiquement détecté que vous alliez traduire en anglais, grâce au nom du fichier qui contient "en". Si la chaîne à traduire peut être mise au pluriel, Qt Linguist vous demandera 2 traductions : une au singulier ("There is %n file") et une au pluriel ("There are %n files").
- **Warnings** : affiche des avertissements bien utiles, comme "Vous avez oublié de mettre un & pour faire un raccourci", ou "La chaîne traduite ne se termine pas par le même signe de ponctuation" (ici un deux-points). Cette zone peut afficher aussi la chaîne à traduire dans son contexte du code source.

C'est maintenant au traducteur de traduire tout ça ! 😊

Lorsqu'il est sûr de sa traduction, il doit marquer la chaîne comme étant validée (en cliquant sur le petit "?" ou en faisant Ctrl +

Entrée). Un petit symbole coché vert doit apparaître, et le dock context doit afficher que toutes les chaînes ont bien été traduites (16/16 par exemple).

Voici les 3 états que peut avoir chaque message :



On procède donc en 2 temps : d'abord on traduit, puis ensuite on se relit et on valide. Lorsque toutes les chaînes sont validées (en vert), le traducteur vous rend le fichier .ts.

Il ne nous reste plus qu'une étape : compiler ce .ts en un .qm, et adapter notre programme pour qu'il charge automatiquement le programme dans la bonne langue.

Lancer l'application traduite

Dernière ligne droite !

Nous avons le .ts entièrement traduit par notre traducteur adoré, il ne nous reste plus qu'à le compiler dans le format final binaire .qm, et à le charger dans l'application.

Compiler le .ts en .qm

Pour effectuer cette compilation, nous devons utiliser un autre programme de Qt : *lrelease*.

Ouvrez donc une console Qt (Qt Command Prompt), rendez-vous dans le dossier de votre projet, et tapez :

Code : Console

```
lrelease nomDuFichier.ts
```

... pour compiler le fichier .ts indiqué.

Vous pouvez aussi faire :

Code : Console

```
lrelease nomDuProjet.pro
```

... pour compiler tous les fichiers .ts du projet.

Comme je viens de terminer la traduction anglaise, je vais compiler le fichier .ts anglais :

Code : Console

```
C:\Users\Mateo\Projets\ZeroClassGenerator>lrelease zeroclassgenerator_en.ts
Updating 'zeroclassgenerator_en.qm'...
Generated 17 translation(s) (17 finished and 0 unfinished)
```

Vous pouvez voir que *lrelease* ne compile que les chaînes marquées comme terminées (celles qui ont le symbole vert dans Qt Linguist). Si certaines ne sont pas marquées comme terminées, elles ne seront pas compilées dans le .qm.



Les chaînes non traduites ou non validées n'apparaîtront donc pas dans le programme. Dans ce cas, c'est la chaîne par défaut écrite dans le code (ici en français) qui sera affichée à la place. D'autre part, notez que vous pouvez aussi faire la même chose directement dans Qt Linguist, en allant dans le menu File / Release.

Nous avons maintenant un fichier `zeroclassgenerator_en.qm` dans le dossier de notre projet. Cool. Si on le chargeait dans notre programme maintenant ? 😊

Charger un fichier de langue .qm dans l'application

Le chargement d'un fichier de langue s'effectue au début de la fonction `main()`. Pour le moment, votre fonction `main()` devrait ressembler à quelque chose comme ceci :

Code : C++

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Juste après la création de l'objet de type `QApplication`, nous allons rajouter les lignes suivantes :

Code : C++

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QTranslator translator;
    translator.load("zeroclassgenerator_en");
    app.installTranslator(&translator);

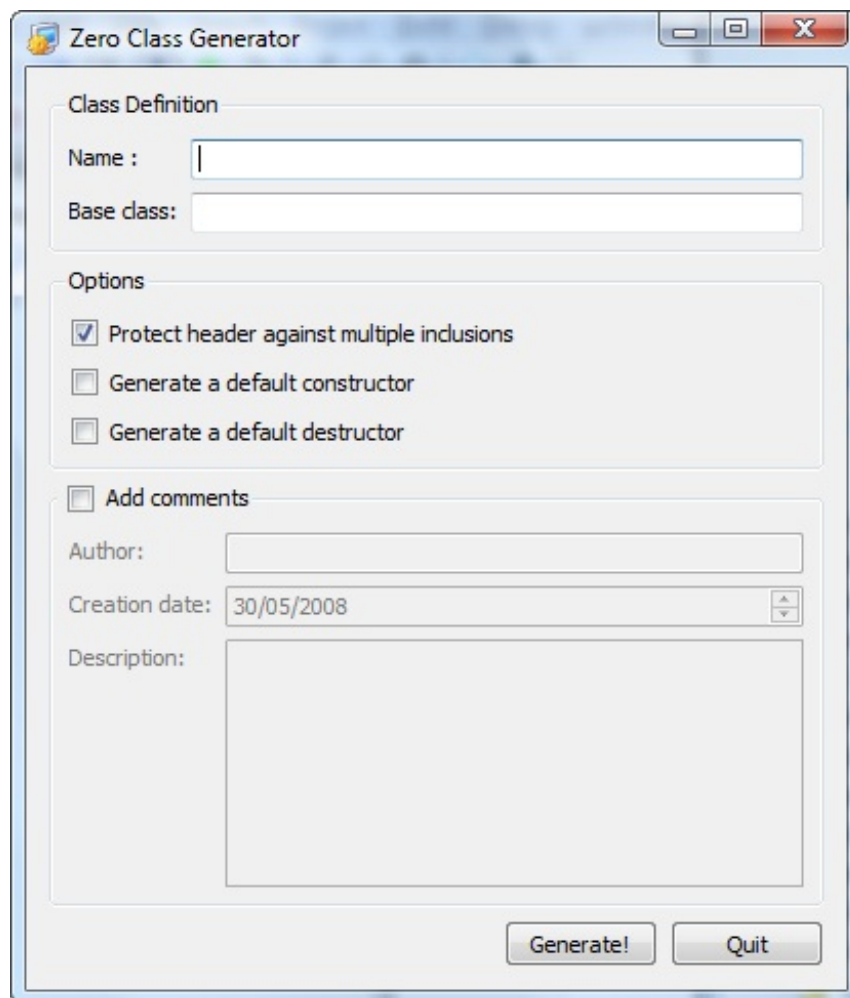
    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```



Vérifiez bien que le fichier `.qm` se trouve dans le même dossier que l'exécutable, sinon la traduction ne sera pas chargée et vous aurez toujours l'application en français !

Si tout va bien, bravo, vous avez traduit votre application ! 😊



Euh, c'est bien mais c'est pas pratique. Là, mon application se chargera toujours en anglais. Il n'y a pas moyen qu'elle s'adapte à la langue de l'utilisateur ? 🤔

Si, bien sûr, c'est faisable. C'est même ce qu'on fera dans 99% des cas.
Dans ce cas, on peut procéder comme ceci :

Code : C++

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);

    QTranslator translator;
    translator.load(QString("zeroclassgenerator_") + locale);
    app.installTranslator(&translator);

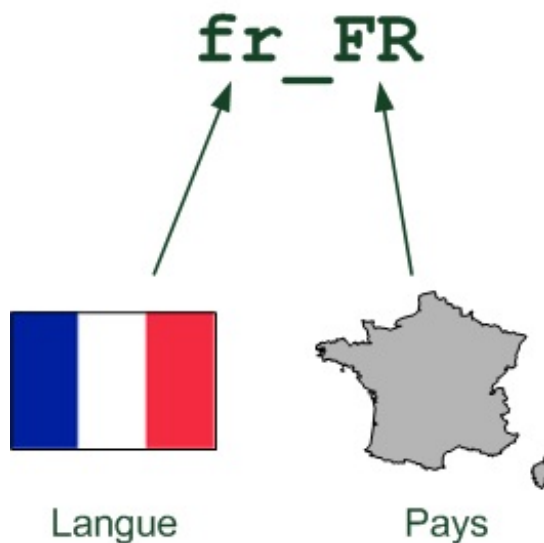
    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Explication : on veut récupérer le code à 2 lettres de la langue du PC de l'utilisateur. On utilise une méthode statique de `QLocale` pour récupérer des informations sur le système d'exploitation sur lequel le programme a été lancé.

La méthode `QLocale::system().name()` renvoie un code ressemblant à ceci : "fr_FR", où "fr" est la langue (français) et "FR" le pays (France).

Si vous êtes québécois, vous aurez par exemple "fr_CA" (français au Canada).



On veut juste récupérer les 2 premières lettres. On utilise la méthode `section()` pour couper la chaîne en deux autour de l'underscore "_". Les 2 autres paramètres permettent d'indiquer qu'on veut le premier mot à gauche de l'underscore, à savoir le "fr".

Au final, notre variable locale contiendra juste ce qu'on veut : la langue de l'utilisateur (par exemple "fr"). On combine cette variable avec le début du nom du fichier de traduction, comme ceci :

Code : C++

```
QString("zeroclassgenerator_") + locale
```

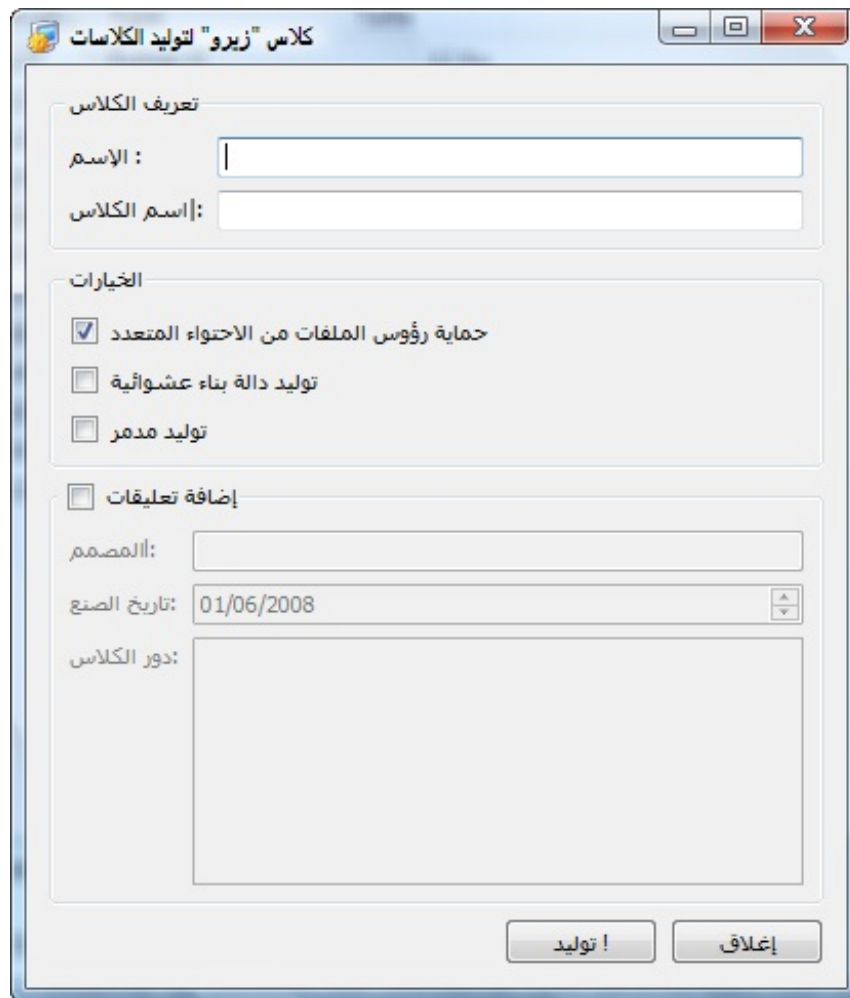
Si locale vaut "fr", le fichier de traduction chargé sera "zeroclassgenerator_fr".
Si locale vaut "en", le fichier de traduction chargé sera "zeroclassgenerator_en".

C'est compris ? 😊

Grâce à ça, notre programme ira chercher le fichier de traduction correspondant à la langue de l'utilisateur. Au pire des cas, si le fichier de traduction n'existe pas car vous n'avez pas fait de traduction dans cette langue, c'est la langue française qui sera utilisée.

Vous voilà maintenant aptes à traduire dans n'importe quelle langue ! 😊

Pour information, voilà ce que donne le ZeroClassGenerator traduit en arabe (merci à *zoro_2009* pour la traduction !) :



Voilà donc la preuve que Qt peut vraiment gérer tous les caractères de la planète grâce à son support de l'Unicode. 😊
Comme vous avez pu le constater, la traduction d'applications Qt est un processus bien rôdé : tout est prévu ! 😊


Vous avez maintenant tous les outils en main pour diffuser votre programme partout dans le monde, même au Japon ! Encore faut-il trouver un traducteur japonais...

... et pour ça, désolé les amis, mais je ne pourrai vraiment pas vous aider. 😞

Modéliser ses fenêtres avec Qt Designer

A force d'écrire le code de vos fenêtres, vous devez peut-être commencer à trouver ça long et répétitif. C'est amusant au début, mais au bout d'un moment on en a un peu marre d'écrire des constructeurs de 3 kilomètres de long juste pour placer les widgets sur la fenêtre.

C'est là que Qt Designer vient vous sauver la vie. Il s'agit d'un programme livré avec Qt (vous l'avez donc déjà installé) qui permet de **dessiner vos fenêtres visuellement**. Mais plus encore, Qt Designer vous permet aussi de modifier les propriétés des widgets, d'utiliser des layouts, et d'effectuer la connexion entre signaux et slots.

 Qt Designer n'est pas un programme magique qui va réfléchir à votre place. Il vous permet juste de gagner du temps et d'éviter les tâches répétitives d'écriture du code de génération de la fenêtre.
N'utilisez PAS Qt Designer et ne lisez PAS ce chapitre si vous ne savez pas coder vos fenêtres à la main. En clair, si vous avez voulu sauter les chapitres précédents et juste lire celui-ci parce que vous le trouvez attirant, vous allez vous planter. C'est dit. 🤔

Nous commencerons par apprendre à manipuler Qt Designer lui-même. Vous verrez que c'est un outil complexe mais qu'on s'y fait vite car il est assez intuitif.

Ensuite, nous apprendrons à utiliser les fenêtres générées avec Qt Designer dans notre code source. Comme vous le verrez, il y a plusieurs façons de faire en fonction de vos besoins.

C'est parti ! 😊

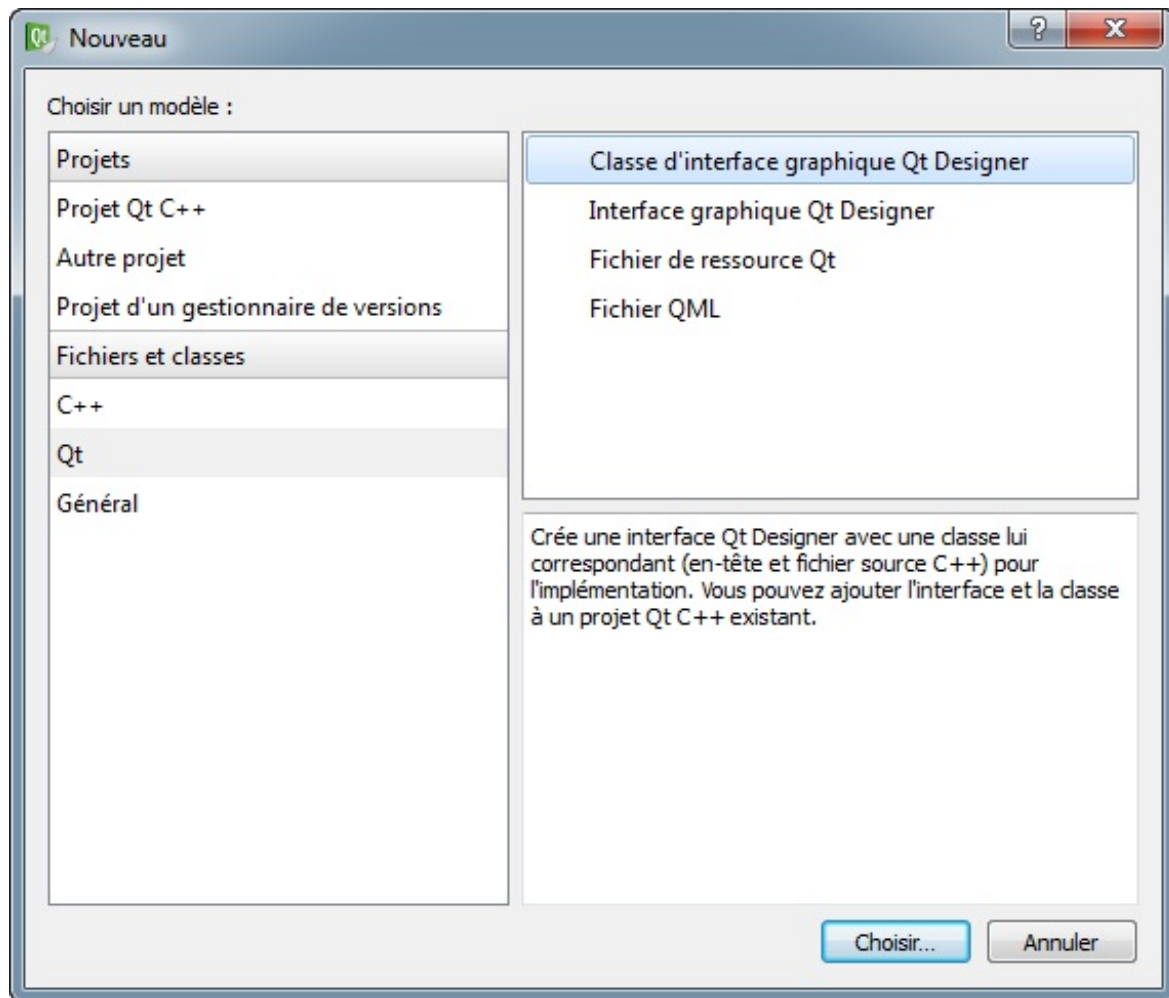
Présentation de Qt Designer



Qt Designer existe sous forme de programme indépendant (cf icône ci-contre), mais il est aussi intégré au sein de Qt Creator dans la section `Design`. Il est plus simple de travailler directement à l'intérieur de Qt Creator, et ça ne change strictement rien aux possibilités qui vous sont offertes. En effet, Qt Designer est réellement intégré dans Qt Creator ! 😊

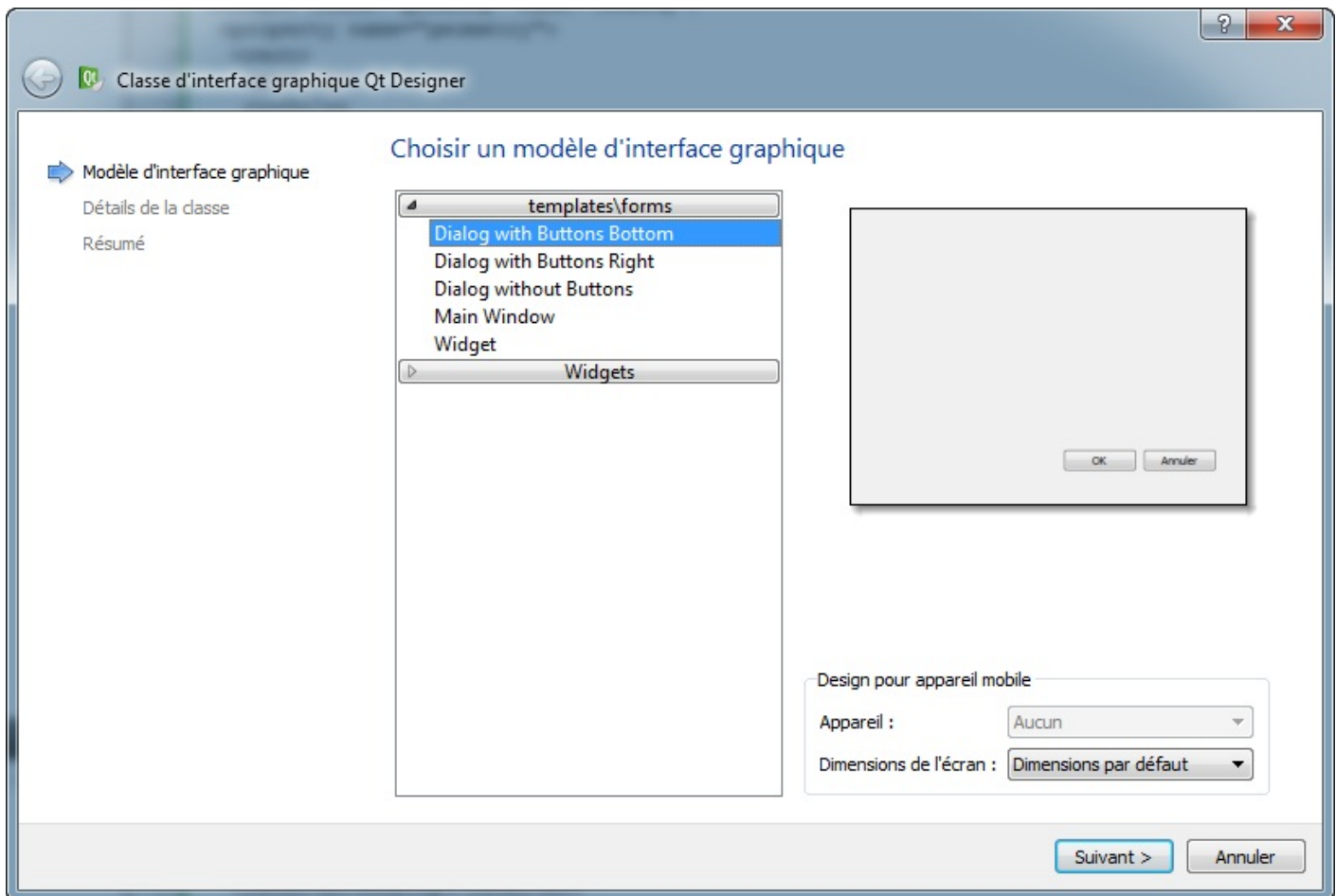
Comme c'est la plus simple et que cette solution n'a que des avantages, nous allons donc travailler directement dans Qt Creator.

Je vais supposer que vous avez déjà créé un projet dans Qt Creator. Pour ajouter une fenêtre de Qt Designer, allez dans le menu `Fichier / Nouveau fichier ou projet` puis sélectionnez `Qt / Classe d'interface graphique Qt Designer`.



Choix du type de fenêtre à créer

Lorsque vous demandez à créer une fenêtre, on vous demande de choisir le type de fenêtre :



Les 3 premiers choix correspondent à des QDialog.

Vous pouvez aussi créer une QMainWindow si vous avez besoin de gérer des menus et des barres d'outils.

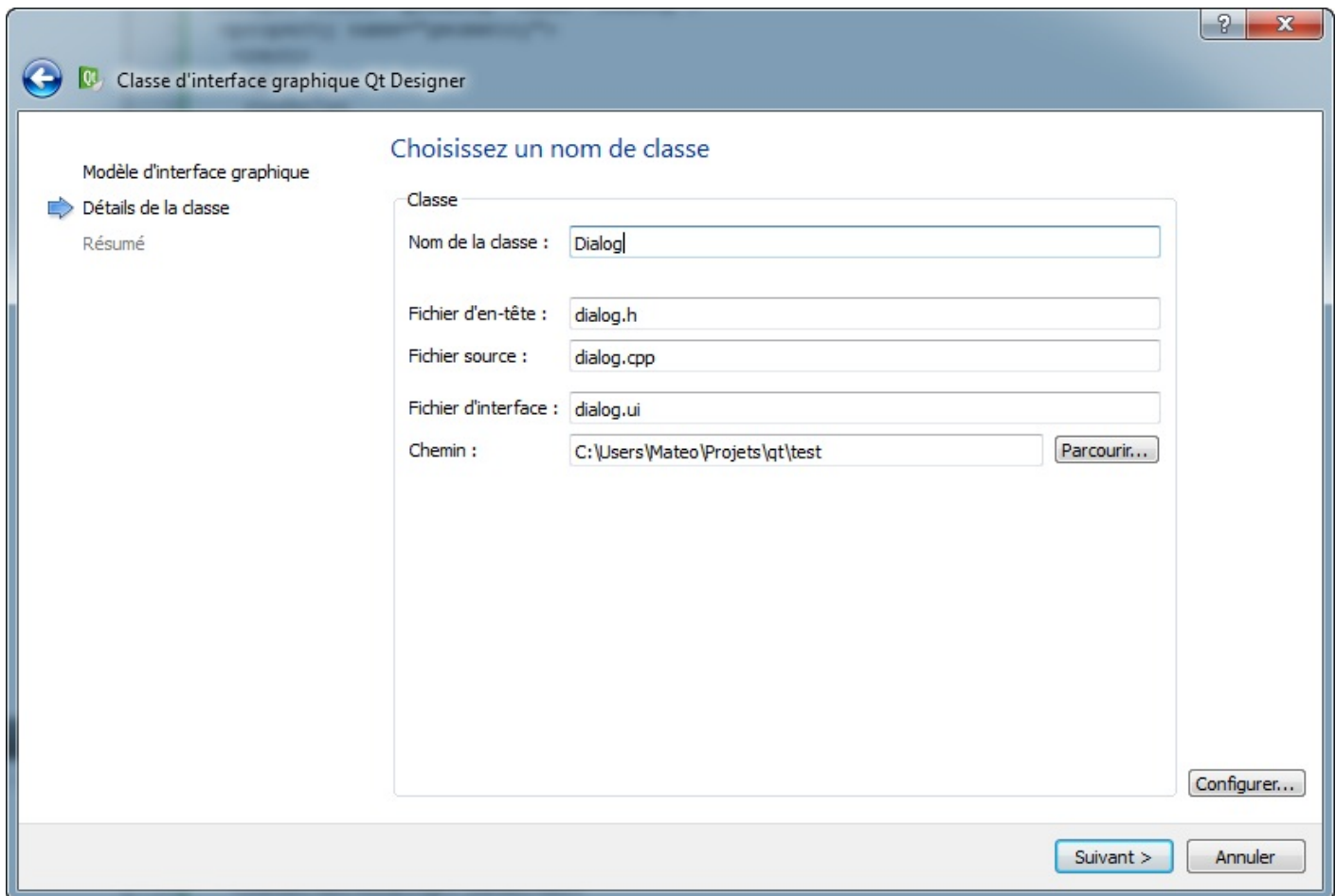
Enfin, le dernier choix correspond à une simple fenêtre de type QWidget.

Pour tester Qt Designer, peu importe le choix que vous ferez ici. On peut partir sur une QDialog si vous voulez (premier choix par exemple).



Il y a d'autres choix que je ne détaillerai pas ici, dans la sous-catégorie "Widgets". Par exemple, on peut créer une fenêtre-QGroupBox. Vous utiliserez très rarement ces choix.

Dans la fenêtre suivante, on vous demande le nom des fichiers à créer. Pour le moment vous pouvez laisser par défaut :

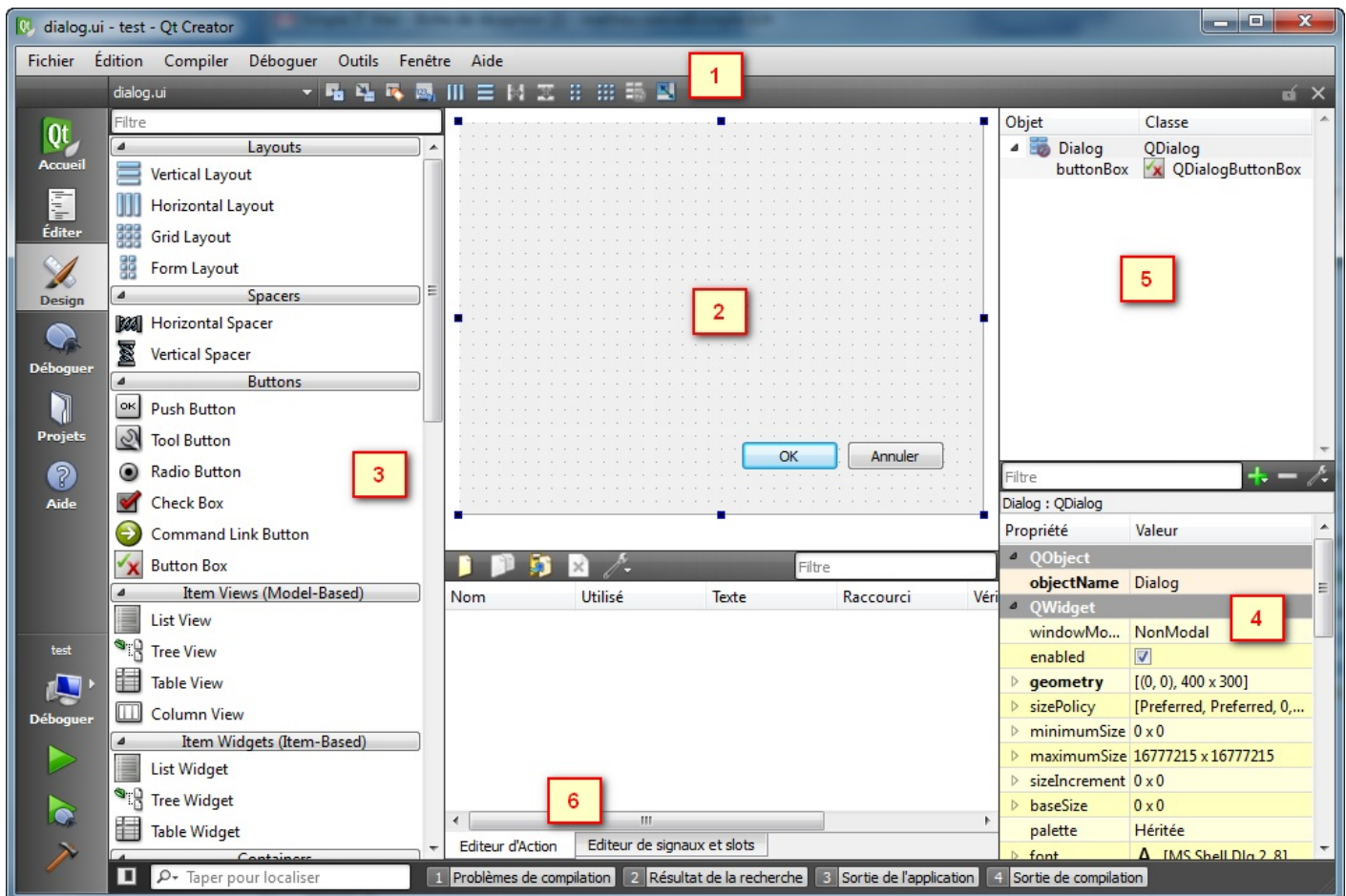


Trois fichiers seront créés :

- dialog.ui : c'est le fichier qui contiendra l'interface graphique (de type XML). C'est ce fichier que nous modifierons avec l'éditeur Qt Designer.
- dialog.h : permet de charger le fichier .ui dans votre projet C++ (en-tête de classe).
- dialog.cpp : permet de charger le fichier .ui dans votre projet C++ (code source de classe).

Analyse de la fenêtre de Qt Designer

Lorsque vous avez créé votre fenêtre, Qt Designer s'ouvre au sein de Qt Creator :



Notez que nous sommes dans la section Design de Qt Creator d'après le menu de gauche. Vous pouvez retrouver les fichiers de votre projet en cliquant sur **Editer**.



Wow ! Mais comment je vais faire pour m'y retrouver avec tous ces boutons ? 🤔

En y allant méthodiquement. 😊

Notez que la position des fenêtres peut être un peu différente chez vous, ne soyez pas surpris. Détaillons chacune des zones importantes dans l'ordre :

1. Sur la **barre d'outils** de Qt Designer, au moins 4 boutons méritent votre attention. Ce sont les 4 boutons situés sous la marque "(1)" rouge que j'ai placée sur la capture d'écran.



Ils permettent de passer d'un mode d'édition à un autre. Qt Designer propose 4 modes d'édition :

- **Editer les widgets** : le mode par défaut, que vous utiliserez le plus souvent. Il permet d'insérer des widgets sur la fenêtre et de modifier leurs propriétés.
- **Editer signaux/slots** : permet de créer des connexions entre les signaux et les slots de vos widgets.
- **Editer les copains** : permet d'associer des QLabel avec leurs champs respectifs. Lorsque vous faites un layout de type QFormLayout, ces associations sont automatiquement créées.
- **Editer l'ordre des onglets** : permet de modifier l'ordre de tabulation entre les champs de la fenêtre, pour ceux qui naviguent au clavier et passent d'un champ à l'autre en appuyant sur la touche "Tab".

Nous ne verrons dans ce chapitre que les 2 premiers modes (Editer les widgets et Editer signaux/slots). Les autres modes sont peu importants et je vous laisse les découvrir par vous-mêmes.

2. Au **centre** de Qt Designer, vous avez la fenêtre que vous êtes en train de dessiner. Pour le moment celle-ci est vide. Si vous créez une QMainWindow, vous aurez en plus une barre de menus et une barre d'outils. Leur édition se fait à la souris, c'est très intuitif. Si vous créez une QDialog, vous aurez probablement des boutons "OK" et "Annuler" déjà disposés.
3. **Widget Box** : ce dock vous donne la possibilité de sélectionner un widget à placer sur la fenêtre. Vous pouvez constater qu'il y a un assez large choix ! Heureusement, ceux-ci sont organisés par groupes pour y voir plus clair. Pour placer un de ces widgets sur la fenêtre, il suffit de faire un glisser-déplacer. Simple et intuitif.
4. **Property Editor** : lorsqu'un widget est sélectionné sur la fenêtre principale, vous pouvez éditer ses propriétés. Vous noterez que les widgets possèdent en général beaucoup de propriétés, et que celles-ci sont organisées en fonction de la classe dans laquelle elles ont été définies. On peut ainsi modifier toutes les propriétés dont un widget hérite, en plus des propriétés qui lui sont propres.



Comme toutes les classes héritent de QObject, vous aurez toujours la propriété `objectName`. C'est le nom de l'objet qui sera créé. N'hésitez pas à le personnaliser, afin d'y voir plus clair tout à l'heure dans votre code source (sinon vous aurez par exemple des boutons appelés `pushButton`, `pushButton_2`, `pushButton_3`, ce qui n'est pas très clair).

Si aucun widget n'est sélectionné, ce sont les propriétés de la fenêtre que vous éditez. Vous pourrez donc par exemple modifier son titre avec la propriété `windowTitle`, son icône avec `windowIcon`, etc.

5. **Object Inspector** : affiche la liste des widgets placés sur la fenêtre, en fonction de leur relation de parenté, sous forme d'arbre. Ça peut être pratique si vous avez une fenêtre complexe et que vous commencez à vous perdre dedans.
6. **Éditeur de signaux/slots et éditeur d'action** : ils sont séparés par des onglets. L'éditeur de signaux/slots est utile si vous avez associé des signaux et des slots, les connexions du widget sélectionné apparaissant ici. Nous verrons comment réaliser des connexions dans Qt Designer tout à l'heure. L'éditeur d'action permet de créer des QAction. C'est donc utile lorsque vous créez une QMainWindow avec des menus et une barre d'outils.

Voilà qui devrait suffire pour une présentation générale de Qt Designer. Maintenant, pratiquons un peu. 😊

Placer des widgets sur la fenêtre

Placer des widgets sur la fenêtre est en fait très simple : vous prenez le widget que vous voulez dans la liste à gauche, et vous le faites glisser où vous voulez sur la fenêtre.

Ce qui est très important à savoir, c'est qu'on peut placer ses widgets de 2 manières différentes :

- **De manière absolue** : vos widgets seront disposés au pixel près sur la fenêtre. C'est la méthode par défaut, la plus précise, mais la moins flexible aussi. Je vous avais parlé de ses défauts dans le chapitre sur les layouts.
- **Avec des layouts** (recommandé pour les fenêtres complexes) : vous pouvez utiliser tous les layouts que vous connaissez. Verticaux, horizontaux, en grille, en formulaire... Grâce à cette technique, les widgets s'adapteront automatiquement à la taille de votre fenêtre.

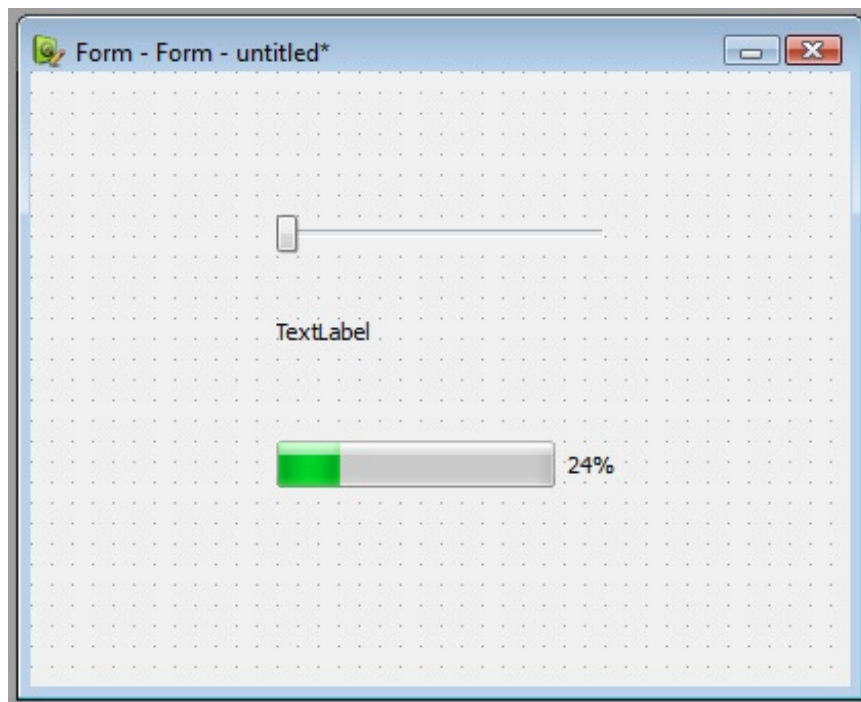
Commençons par les placer de manière absolue, puis nous verrons comment utiliser les layouts dans Qt Designer.

Placer les widgets de manière absolue

Je vous propose pour vous entraîner de faire une petite fenêtre simple composée de 3 widgets :

- QSlider
- QLabel
- QProgressBar

Votre fenêtre devrait à peu près ressembler à ceci maintenant :



Vous pouvez déplacer ces widgets comme bon vous semble sur la fenêtre.
Vous pouvez les agrandir ou les rétrécir.

Quelques raccourcis à connaître :

- En maintenant la touche **Ctrl** appuyée, vous pouvez sélectionner plusieurs widgets en même temps.
- Faites **Suppr** pour supprimer les widgets sélectionnés.
- Si vous maintenez la touche **Ctrl** enfoncée lorsque vous déplacez un widget, celui-ci sera copié.
- Vous pouvez **double-cliquer** sur un widget pour modifier son nom (il vaut mieux donner un nom personnalisé plutôt que laisser le nom par défaut).
Sur certains widgets complexes, comme la QComboBox (liste déroulante), le double clic a pour effet de vous permettre d'éditer la liste des éléments contenus dans la liste déroulante.
- Pensez aussi à faire un **clic droit** sur les widgets pour modifier certaines propriétés, comme la bulle d'aide (toolTip).

Utiliser les layouts

Pour le moment, nous n'utilisons aucun layout. Si vous essayez de redimensionner la fenêtre, vous verrez que les widgets ne s'adaptent pas à la nouvelle taille et qu'ils peuvent même disparaître si on réduit trop la taille de la fenêtre !

Il y a 2 façons d'utiliser des layouts :

- Utiliser la barre d'outils en haut.
- Glisser-déplacer des layouts depuis le dock de sélection de widgets ("Widget Box").

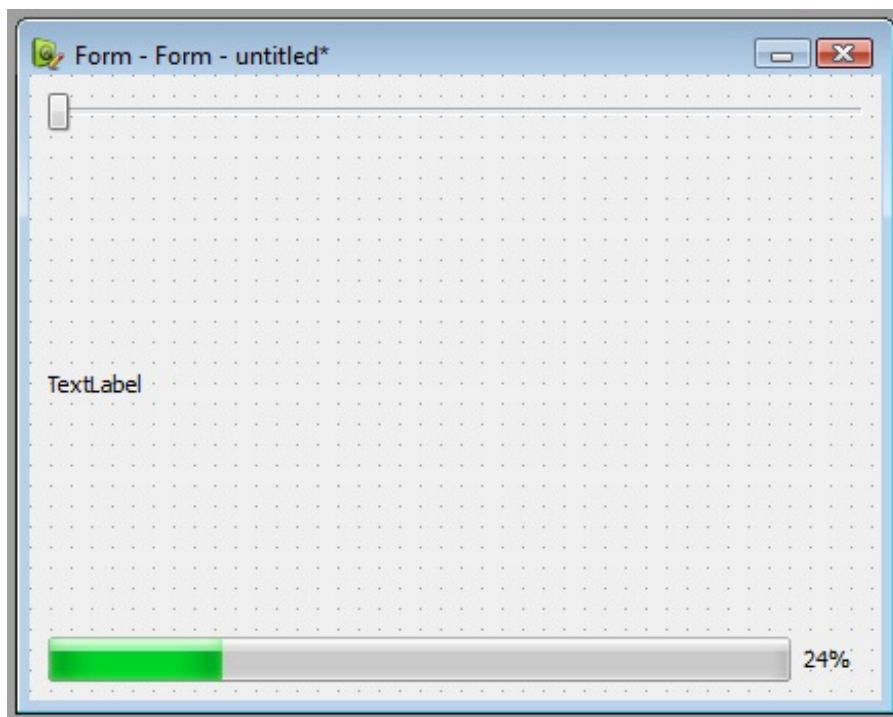
Pour une fenêtre simple comme celle-là, nous n'aurons besoin que d'un layout principal.
Pour définir ce layout principal, le mieux est de passer par la barre d'outils :



Cliquez sur une zone vide de la fenêtre (en clair, il faut que ce soit la fenêtre qui soit sélectionnée et non un de ses widgets). Vous devriez alors voir les boutons de la barre d'outils des layouts s'activer, comme sur l'image ci-dessus.

Cliquez sur le bouton correspondant au layout vertical (le second) pour organiser automatiquement la fenêtre selon un layout vertical. 😊

Vous devriez alors voir vos widgets s'organiser comme ceci :



C'est le layout vertical qui les place comme ça afin qu'ils occupent toute la taille de la fenêtre. Bien sûr, vous pouvez réduire la taille de la fenêtre si vous le désirez.

Vous pouvez aussi demander à ce que la fenêtre soit réduite à la taille minimale acceptable, en cliquant sur le bouton tout à droite de la barre d'outils, intitulé "Adjust Size".



Maintenant que vous avez défini le layout principal de la fenêtre, sachez que vous pouvez insérer un sous-layout en plaçant par exemple un des layouts proposés dans la Widget Box.

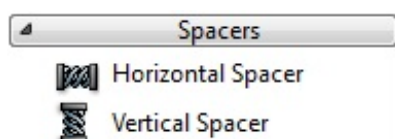
Insérer des spacers

Vous trouvez que la fenêtre est un peu moche si on l'agrandit trop ?

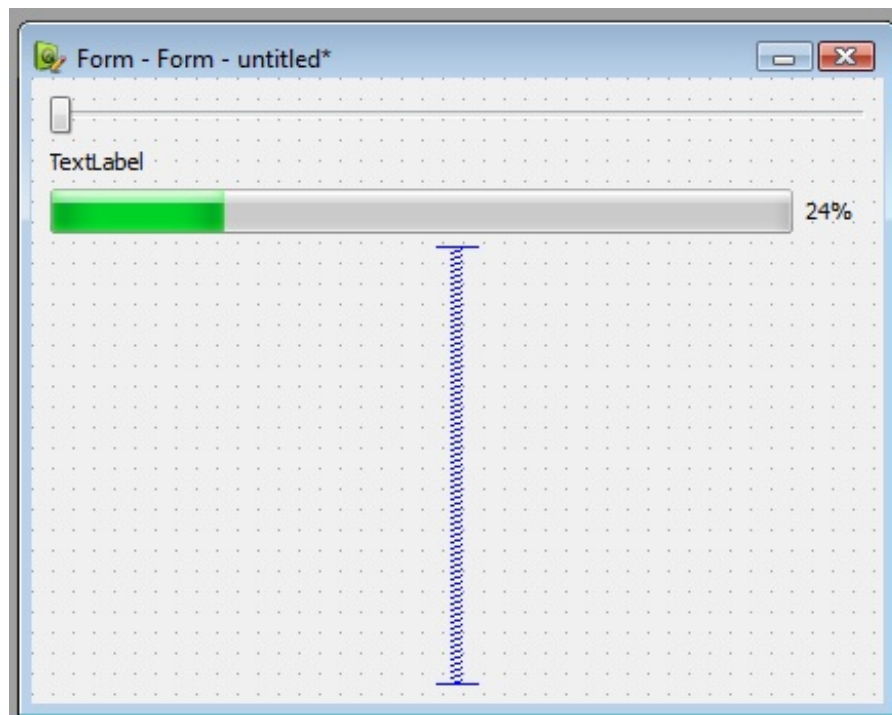
Moi aussi. Les widgets sont trop espacés, ça ne me convient pas.

Pour changer la position des widgets *tout en conservant le layout*, on peut insérer un spacer. Il s'agit d'un widget invisible qui sert à créer de l'espace sur la fenêtre.

Le mieux est encore d'essayer pour comprendre ce que ça fait. Dans la Widget Box, vous devriez avoir une section "Spacers" :



Prenez un "Vertical Spacer", et insérez-le tout en bas de la fenêtre. Vous devriez alors voir ceci :



Le spacer va forcer les autres widgets à se coller tout en haut. Ils sont toujours organisés selon un layout, mais au moins maintenant nos widgets sont plus rapprochés les uns des autres.

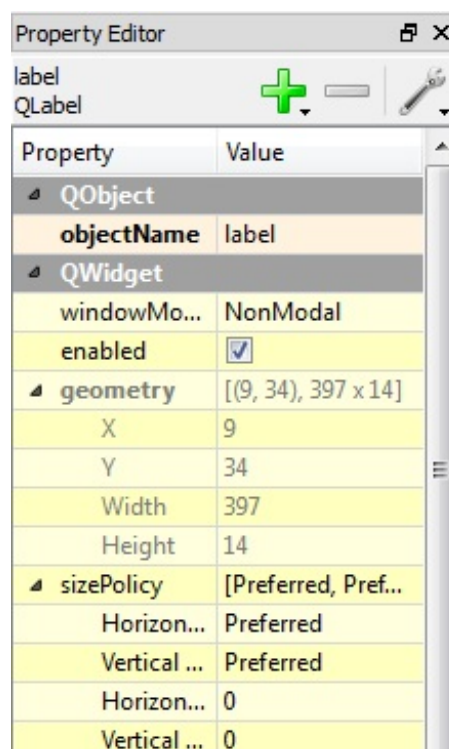
Essayez de déplacer le spacer sur la fenêtre pour voir. Placez-le entre le libellé et la barre de progression. Vous devriez voir que la barre de progression se colle maintenant tout en bas.

Le comportement du spacer est assez logique, mais il faut l'essayer pour bien comprendre. 😊

Editer les propriétés des widgets

Il nous reste une chose très importante à voir : l'édition des propriétés des widgets.

Sélectionnez par exemple le libellé (QLabel). Regardez le dock intitulé "Property Editor". Il affiche maintenant les propriétés du QLabel :



Ces propriétés sont organisées en fonction de la classe dans laquelle elles ont été définies, et c'est une bonne chose. Je m'explique.

Vous savez peut-être qu'un QLabel hérite de QFrame, qui hérite de QWidget, qui hérite lui-même de QObject ?

Chacune de ces classes définit des propriétés. QLabel hérite donc des propriétés de QFrame, QWidget et QObject, mais a aussi des propriétés qui lui sont propres.

Sur ma capture d'écran ci-dessus, on peut voir une propriété de QObject : `objectName`. C'est le nom de l'objet qui sera créé dans le code. Je vous conseille de le personnaliser pour que vous puissiez vous y retrouver dans le code source ensuite.



La plupart du temps, on peut éditer le nom d'un widget en double-cliquant dessus sur la fenêtre.

Si vous descendez un peu plus bas dans la liste, vous devriez vous rendre compte qu'un grand nombre de propriétés sont proposées par QWidget (notamment la police, le style de curseur de la souris, etc.). Descendez encore plus bas. Vous devriez arriver sur les propriétés héritées de QFrame, puis celles propres à QLabel :

styleSheet	
▶ locale	French, France
▾ QFrame	
frameShape	NoFrame
frameShadow	Plain
lineWidth	1
midLineWid...	0
▾ QLabel	
▶ text	TextLabel
textFormat	AutoText
pixmap	
scaledCont...	<input type="checkbox"/>
▶ alignment	AlignLeft, Align...
wordWrap	<input type="checkbox"/>
margin	0
indent	-1
openExtern...	<input type="checkbox"/>
▶ textInteracti...	LinksAccessible...
buddy	

Comme vous pouvez le voir, ces propriétés ont été mises en valeur : elles sont en vert.

Je trouve que c'est très bien d'avoir organisé les propriétés comme ça. Ainsi, on voit bien où elles sont définies.

Vous devriez modifier la propriété `text`, pour changer le texte affiché dans le QLabel. Mettez par exemple "0". Amusez-vous à changer la police (propriété `font` issue de QWidget) ou encore à mettre une bordure (propriété `frameShape` issue de QFrame).

Vous remarquerez que lorsque vous éditez une propriété, son nom s'affiche en gras pour être mis en valeur. Cela vous permet par la suite de repérer du premier coup d'oeil les propriétés que vous avez modifiées.



Certaines propriétés, comme `alignement` de QLabel, possèdent des sous-propriétés. Cliquez sur la petite flèche à gauche pour afficher et modifier ces sous-propriétés. Essayez de faire en sorte que le texte de notre libellé soit centré horizontalement par exemple.

Modifiez aussi les propriétés de la QProgressBar pour qu'elle affiche 0% pour défaut (propriété `value`).

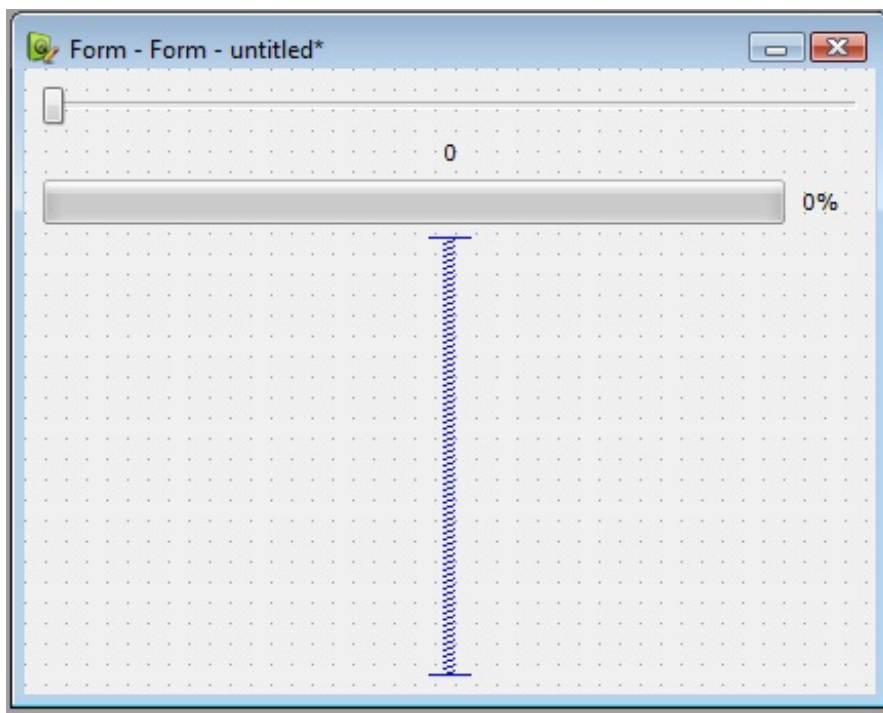
Vous pouvez aussi modifier les propriétés de la fenêtre. Cliquez sur une zone vide de la fenêtre afin qu'aucun widget ne soit

sélectionné. Le dock "Property Editor" vous affichera alors les propriétés de la fenêtre (ici, notre fenêtre est un QWidget, donc vous aurez juste les propriétés de QWidget).



Astuce : si vous ne comprenez pas à quoi sert une propriété, cliquez dessus puis appuyez sur la touche F1. Qt Designer lancera automatiquement Qt Assistant pour afficher l'aide sur la propriété sélectionnée.

Essayez d'avoir une fenêtre qui ressemble au final grosso modo à la mienne :



Le libellé et la barre de progression doivent afficher 0 par défaut.

Bravo, vous savez maintenant insérer des widgets, les organiser selon un layout et personnaliser leurs propriétés dans Qt Designer ! 😊

Nous n'avons utilisé pour le moment que le mode "Edit Widgets". Il nous reste à étudier le mode "Edit Signals/Slots"...

Configurer les signaux et les slots

Passez en mode "Edit Signals/Slots" en cliquant sur le second bouton de la barre d'outils :

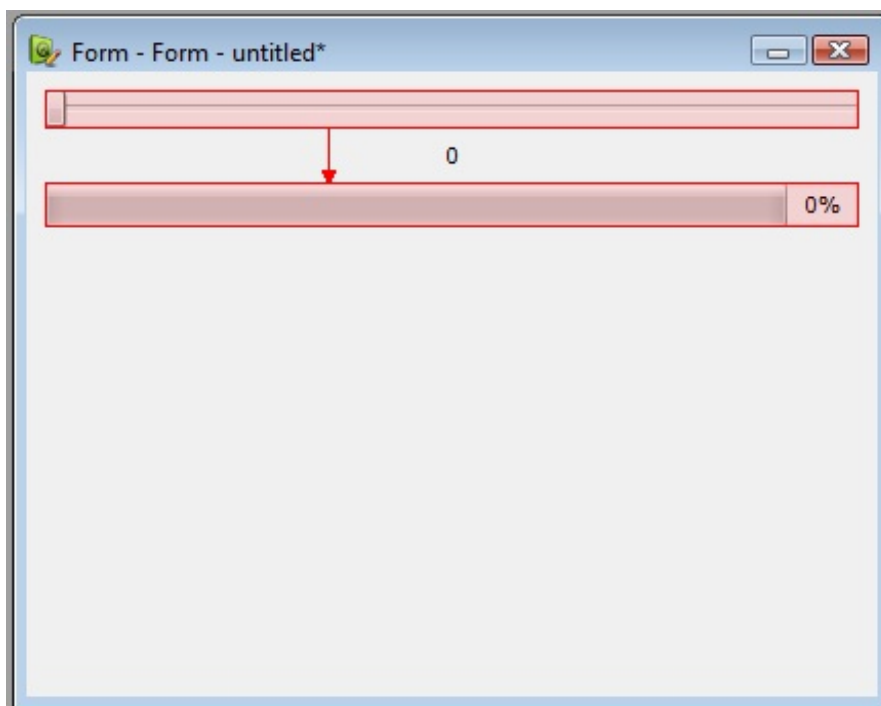


Vous pouvez aussi appuyer sur la touche F4. Vous pourrez faire F3 pour revenir au mode d'édition des widgets.

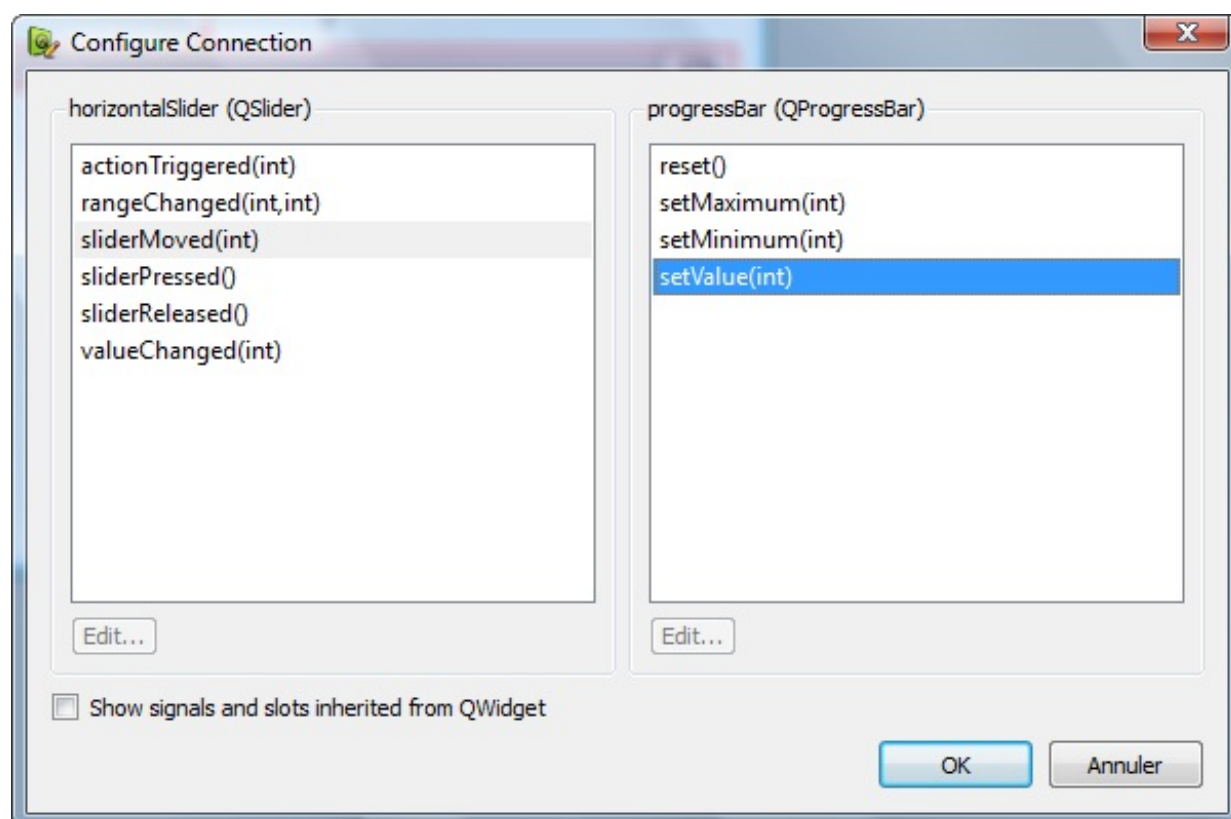
Dans ce mode, on ne peut pas ajouter, modifier, supprimer, ni déplacer de widgets. Par contre, si vous pointez sur les widgets de votre fenêtre, vous devriez voir un cadre rouge autour d'eux.

Vous pouvez, de manière très intuitive, associer les widgets entre eux pour créer des connexions simples entre leurs signaux et slots. Je vous propose par exemple d'associer le QSlider avec notre QProgressBar.

Pour cela, cliquez sur le QSlider et maintenez le bouton gauche de la souris enfoncé. Pointez sur la QProgressBar et relâchez le bouton. La connexion que vous allez faire devrait ressembler à ceci :



Une fenêtre apparaît alors pour que vous puissiez choisir le signal et le slot à connecter :



A gauche : les signaux disponibles dans le QSlider.

A droite : les slots *compatibles* disponibles dans la QProgressBar.

Sélectionnez un signal à gauche, par exemple `sliderMoved(int)`. Ce signal est envoyé dès que l'on déplace un peu le slider. Vous verrez que la liste des slots compatibles apparaît à droite.

En fonction du signal choisi, Qt Designer ne vous affiche que les slots de destination compatibles. Par exemple, `sliderMoved(int)` s'accorde bien avec `setValue(int)`. On peut aussi le connecter à `reset()`, dans ce cas le nombre envoyé en paramètre sera perdu.

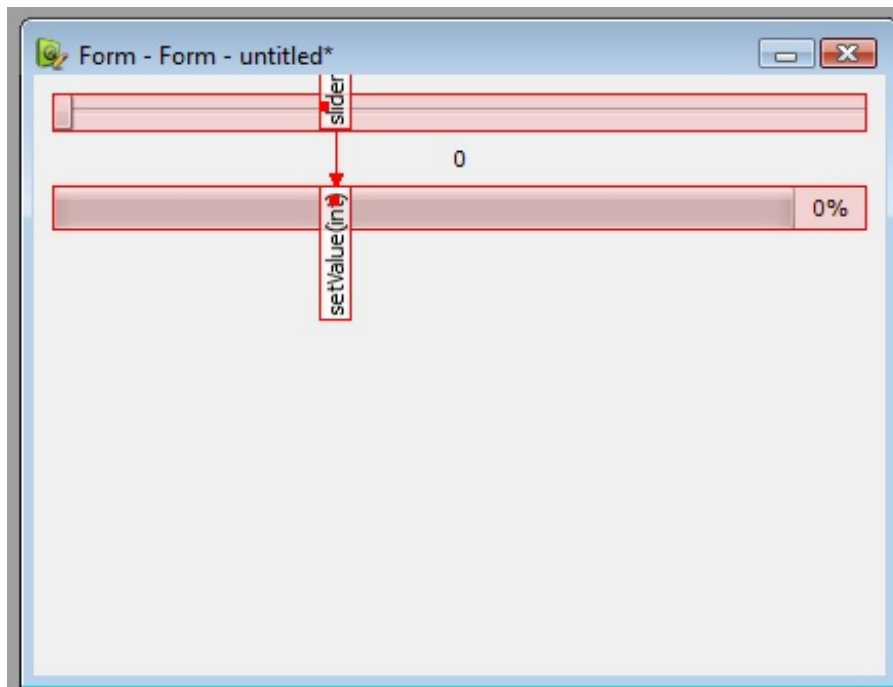
Par contre, on ne peut pas connecter le signal `sliderMoved(int)` au slot `setRange(int, int)` car le signal n'envoie pas



assez de paramètres. D'ailleurs, vous ne devriez pas voir ce slot disponible dans la liste des slots si vous avez choisi le signal `sliderMoved(int)`, ce qui vous empêche de créer une connexion incompatible.

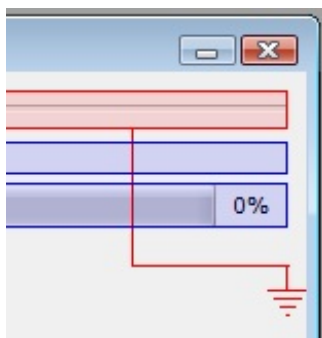
Nous allons connecter `sliderMoved(int)` du `QSlider` avec `setValue(int)` de la `QProgressBar`.

Faites OK pour valider une fois le signal et le slot choisis. C'est bon, la connexion est créée. 😊



Faites de même pour associer `sliderMoved(int)` du `QSlider` à `setNum(int)` du `QLabel`.

Notez que vous pouvez aussi connecter un widget à la fenêtre. Dans ce cas, visez une zone vide de la fenêtre. La flèche devrait se transformer en symbole de masse (bien connu par ceux qui font de l'électricité ou de l'électronique) :



Cela vous permet d'associer un signal du widget à un slot de la fenêtre, ce qui peut vous être utile si vous voulez créer un bouton "Fermer la fenêtre" par exemple.



Attention : si dans la fenêtre du choix du signal et du slot vous ne voyez aucun slot s'afficher pour la fenêtre, c'est normal. Qt les masque par défaut car ils sont nombreux. Si on les affichait pour chaque connexion entre 2 widgets, on en aurait beaucoup trop (puisque tous les widgets héritent de `QWidget`). Pour afficher quand même les signaux et slots issus de `QWidget`, cochez la case "*Show signals and slots inherited from QWidget*".

Pour des connexions simples entre les signaux et les slots des widgets, Qt Designer est donc très intuitif et convient parfaitement.



Eh, mais si je veux créer un slot personnalisé pour faire des manipulations un peu plus complexes, comment je fais ?



Qt Designer ne peut pas vous aider pour ça. Si vous voulez créer un signal ou un slot personnalisé, il faudra le faire tout à l'heure dans le code source (en modifiant les fichiers .h et .cpp qui ont été créés en même temps que le .ui). Comme vous pourrez le voir néanmoins, c'est très simple à faire.

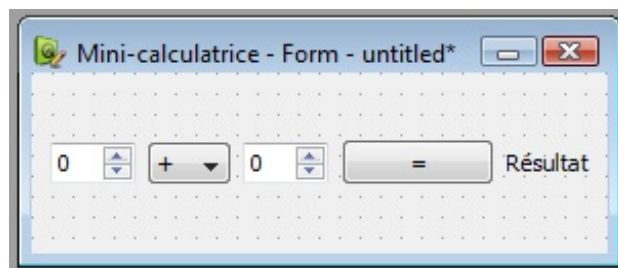
En y réfléchissant bien, c'est même d'ailleurs la seule chose que vous aurez à coder ! En effet, tout le reste est automatiquement géré par Qt Designer. Vous n'avez plus qu'à vous concentrer sur la partie "réflexion" de votre code source. Qt Designer vous permet donc de gagner du temps en vous épargnant les tâches répétitives et basiques qu'on fait à chaque fois que l'on crée une fenêtre.

Utiliser la fenêtre dans votre application

Il reste une dernière étape, et pas des moindres : apprendre à utiliser la fenêtre ainsi créée dans votre application.

Notre nouvel exemple

Je vous propose de créer une nouvelle fenêtre (parce que l'exemple de tout à l'heure était bien joli, mais pas très intéressant à part pour tester les signaux et slots 🤖). On va créer une mini-calculatrice :



Essayez de reproduire à peu près la même fenêtre que moi, de type Widget. Un layout principal horizontal suffira à organiser les widgets.

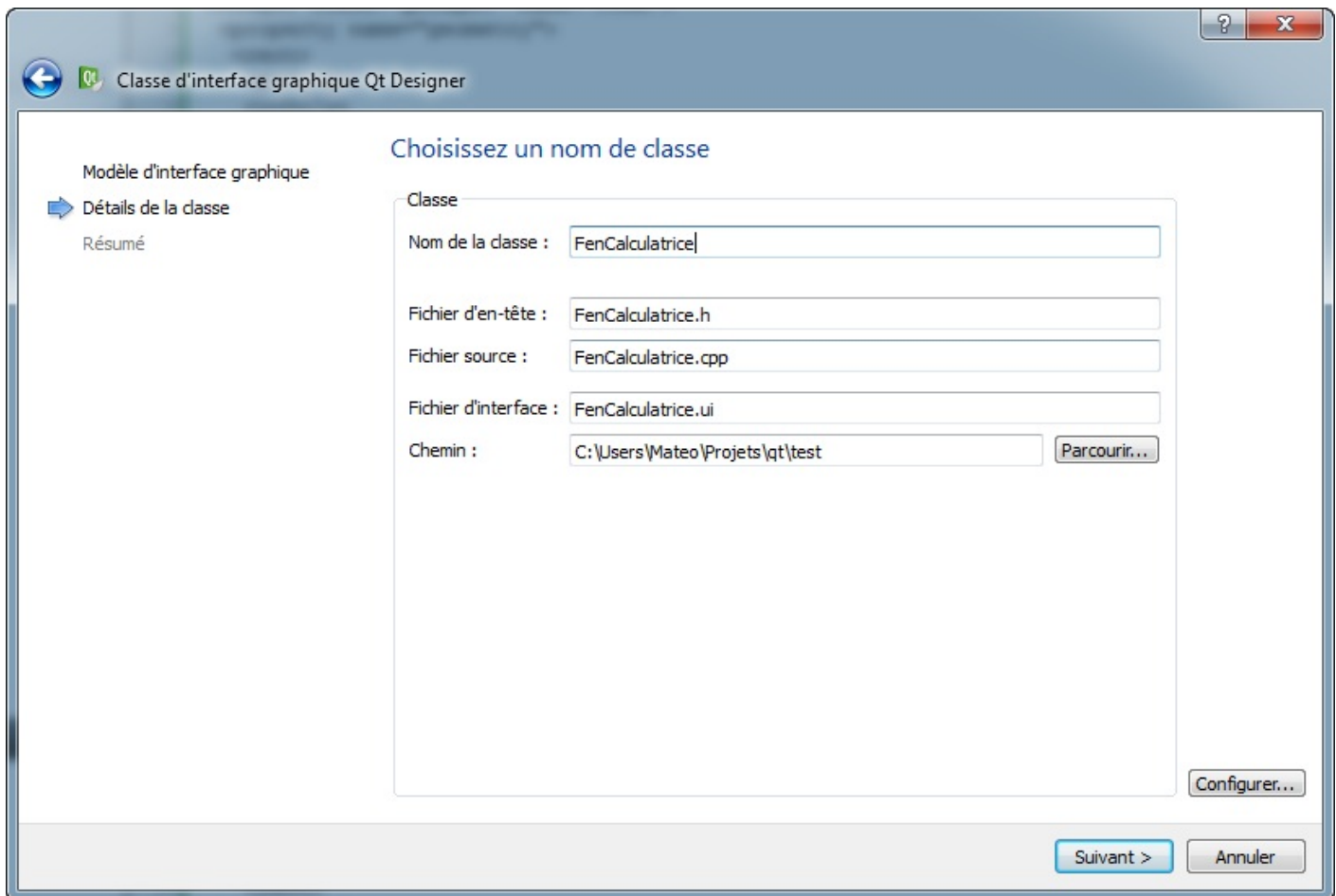
La fenêtre est constituée des widgets suivants, de gauche à droite :

Widget	Nom de l'objet
QSpinBox	nombre1
QComboBox	operation
QSpinBox	nombre2
QPushButton	boutonEgal
QLabel	resultat

Pensez à bien renommer les widgets afin que vous puissiez vous y retrouver dans votre code source ensuite. 🤖

Pour la liste déroulante du choix de l'opération, je l'ai déjà pré-remplie avec 4 valeurs : +, -, * et /. Double-cliquez sur la liste déroulante pour ajouter / supprimer des valeurs.

Il faudra donner un nom à la fenêtre lorsque vous la créerez dans Qt Creator. Je l'ai appelée "**FenCalculatrice**" (de même que les fichiers qui seront créés) :



Le principe de la génération du code source

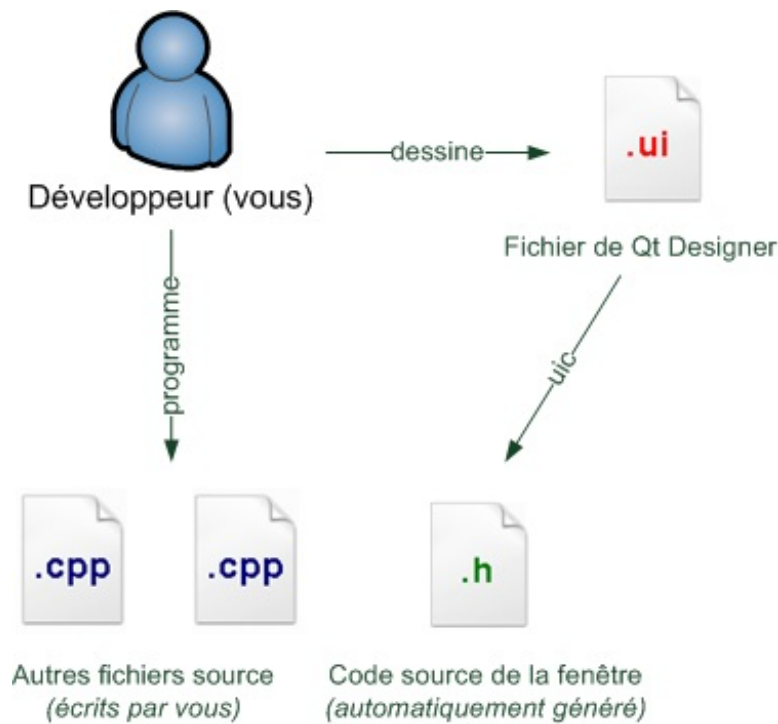
Essayons maintenant de récupérer le code de la fenêtre dans notre application et d'ouvrir cette fenêtre.



Le code ? Quel code ? Je ne vois pas de code moi ?
Qt Designer est censé générer un code source ?

Non, Qt Designer ne fait que produire un fichier .ui. C'est le petit programme *uic* qui se charge de transformer le .ui en code source C++.

Voilà ce que ça donne schématiquement :



Vous dessinez la fenêtre avec Qt Designer qui produit un fichier .ui.

Ce fichier est transformé automatiquement en code source par le petit programme en ligne de commande *uic*. Celui-ci génère un fichier *ui_nomDeVotreFenetre.h*. Qt met tout le code dans le fichier .h, ne vous étonnez donc pas s'il n'y a pas de .cpp correspondant.

Vous continuez à programmer vos autres fichiers source comme avant (.cpp et .h).

A la compilation, le fichier *ui_nomDeVotreFenetre.h* sera compilé avec vos autres fichiers source !



Vous n'appellerez pas *uic* directement, c'est Qt qui le fera pour vous avant la compilation. Ce que je viens de vous expliquer vous permet de mieux comprendre le fonctionnement de Qt, mais en pratique tout cela est transparent pour vous !

Utiliser la fenêtre dans notre application

Pour utiliser la fenêtre créée à l'aide de Qt Designer dans notre application, plusieurs méthodes s'offrent à nous. Le plus simple est encore de laisser Qt Creator nous guider !

Eh oui, souvenez-vous : Qt Creator a créé un fichier .ui, mais aussi des fichiers .cpp et .h de classe ! Ce sont ces derniers fichiers qui vont appeler la fenêtre que nous avons créée.

En pratique, dans la déclaration de la classe générée par Qt Creator (fichier **FenCalculatrice.h**), on retrouve le code suivant :

Code : C++

```

#ifndef FENCALCULATRICE_H
#define FENCALCULATRICE_H

#include <QWidget>

namespace Ui {
    class FenCalculatrice;
}

class FenCalculatrice : public QWidget
{
    Q_OBJECT

```

```

public:
    explicit FenCalculatrice(QWidget *parent = 0);
    ~FenCalculatrice();

private:
    Ui::FenCalculatrice *ui;
};

#endif // FENCALCULATRICE_H

```

Le fichier FenCalculatrice.cpp, lui, contient le code suivant :

Code : C++

```

#include "FenCalculatrice.h"
#include "ui_FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::FenCalculatrice)
{
    ui->setupUi(this);
}

FenCalculatrice::~FenCalculatrice()
{
    delete ui;
}

```



Comment ça marche tout ce bazar ? 🤔

Vous avez une classe FenCalculatrice qui a été créée automatiquement par Qt Creator (fichiers FenCalculatrice.h et FenCalculatrice.cpp). Lorsque vous créez une nouvelle instance de cette classe, la fenêtre que vous avez dessinée tout à l'heure s'affiche !



Pourquoi ? Le fichier de la classe est tout petit et ne fait pas grand chose pourtant ? 🤔

Si, regardez bien :

- Le fichier automatiquement généré par uic a été automatiquement inclus dans le .cpp : `#include "ui_FenCalculatrice.h"`
- Le constructeur charge l'interface définie dans ce fichier auto-généré grâce à `ui->setupUi(this)` ; . C'est cette ligne qui lance la construction de la fenêtre.

Bien sûr, la fenêtre est encore une coquille vide : elle ne fait rien. Utilisez la classe FenCalculatrice pour compléter ses fonctionnalités et la rendre intelligente. Par exemple, dans le constructeur, pour modifier un élément de la fenêtre, vous pouvez faire ceci :

Code : C++

```

FenCalculatrice::FenCalculatrice(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::FenCalculatrice)

```

```
{
    ui->setupUi (this);

    ui->boutonEgal->setText ("Egal");
}
```



Le nom du bouton "boutonEgal", nous l'avons défini dans Qt Designer tout à l'heure (propriété `objectName` de `QObject`). Retournez voir le petit tableau un peu plus haut pour vous souvenir de la liste des noms des widgets de la fenêtre.

Bon en général vous n'aurez pas besoin de personnaliser vos widgets, vu que vous avez tout fait sous Qt Designer. Mais si vous avez besoin d'adapter leur contenu à l'exécution (pour afficher le nom de l'utilisateur par exemple), il faudra passer par là.

Maintenant ce qui est intéressant surtout, c'est d'effectuer une connexion :

Code : C++

```
FenCalculatrice::FenCalculatrice(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::FenCalculatrice)
{
    ui->setupUi (this);

    connect(ui->boutonEgal, SIGNAL(clicked()), this,
    SLOT(calculerOperation()));
}
```



N'oubliez pas à chaque fois de mettre le préfixe "ui" devant chaque nom de widget !

Ce code nous permet de faire en sorte que le slot `calculerOperation()` de la fenêtre soit appelé à chaque fois que l'on clique sur le bouton. Bien sûr, c'est à vous d'écrire le slot `calculerOperation()`.

Il ne vous reste plus qu'à adapter votre main pour appeler la fenêtre comme une fenêtre classique :

Code : C++

```
#include <QApplication>
#include <QtGui>
#include "FenCalculatrice.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    FenCalculatrice fenetre;
    fenetre.show();

    return app.exec();
}
```

Personnaliser le code et utiliser les Auto-Connect

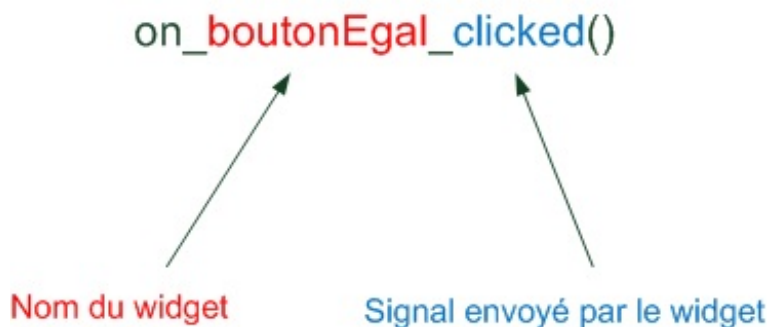
Les fenêtres créées avec Qt Designer bénéficient du système "Auto-Connect" de Qt. C'est un système qui crée les connexions tout seul.

Par quelle magie ?

Il vous suffit en fait de créer des slots en leur donnant un nom qui respecte une convention.

Prenons le widget **boutonEgal** et son signal **clicked()**. Si vous créez un slot appelé `on_boutonEgal_clicked()` dans votre fenêtre, ce slot sera automatiquement appelé lors d'un clic sur le bouton.

La convention à respecter est représentée sur le schéma ci-dessous :



Essayons d'utiliser l'Auto-Connect dans notre programme. Voici le .h qui déclare le slot :

Code : C++

```
#ifndef FENCALCULATRICE_H
#define FENCALCULATRICE_H

#include <QWidget>

namespace Ui {
    class FenCalculatrice;
}

class FenCalculatrice : public QWidget
{
    Q_OBJECT

public:
    explicit FenCalculatrice(QWidget *parent = 0);
    ~FenCalculatrice();

private slots:
    void on_boutonEgal_clicked();

private:
    Ui::FenCalculatrice *ui;
};

#endif // FENCALCULATRICE_H
```

Et voici le .cpp :

Code : C++

```
#include "FenCalculatrice.h"
#include "ui_FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) :
```

```

        QWidget(parent),
        ui(new Ui::FenCalculatrice)
    {
        ui->setupUi(this);
    }

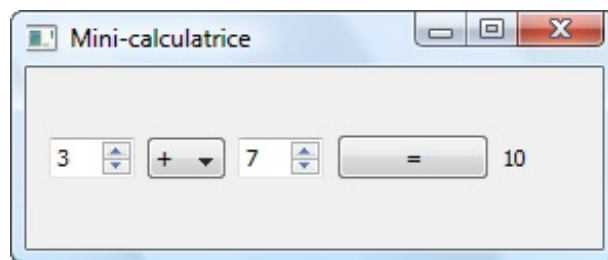
void FenCalculatrice::on_boutonEgal_clicked()
{
    int somme = ui->nombre1->value() + ui->nombre2->value();
    ui->resultat->setNum(somme);
}

FenCalculatrice::~FenCalculatrice()
{
    delete ui;
}

```

Vous noterez qu'on n'a plus besoin de faire de connexion dans le constructeur. Ben oui, c'est le principe de l'Auto-Connect. 😊
Comme vous le voyez, il suffit de créer un slot avec un nom particulier, et tout roule comme sur des roulettes !

Vous pouvez tester le programme, ça marche !



Bon, j'avoue, je n'ai géré ici que l'addition. Mais je vais pas tout vous faire non plus hein. 😊

Exercice (me dites pas que vous l'avez pas vu venir 😊) : complétez le code de la calculatrice pour effectuer la bonne opération en fonction de l'élément sélectionné dans la liste déroulante.



L'Auto-Connect est activé par défaut dans les fenêtres créées avec Qt Designer, mais vous pouvez aussi vous en servir dans vos autres fenêtres "faites main".

Il suffira d'ajouter la ligne suivante dans le constructeur de la fenêtre pour bénéficier de toute la puissance de l'Auto-Connect : `QMetaObject::connectSlotsByName(this);`

Ceux qui croyaient que Qt Designer était un "programme magique qui allait réaliser des fenêtres tout seul sans avoir besoin de coder" en ont été pour leurs frais ! 😊

Pourtant, comme avec Qt Linguist, le processus de création de fenêtres de Qt Designer a été très bien pensé. Tout est logique et s'enchaîne de bout en bout, mais encore faut-il comprendre cette logique. J'espère vous y avoir aidé à travers ce chapitre.

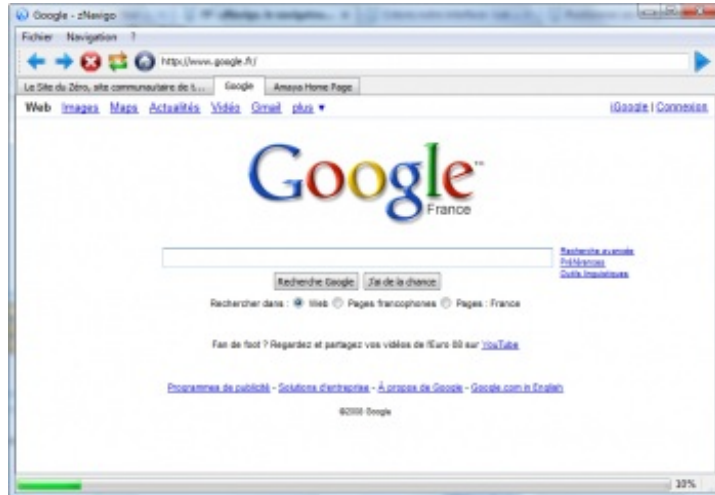
Entraînez-vous à utiliser quelques fenêtres créées avec Qt Designer, et en particulier à créer des slots personnalisés.

Tant qu'à faire, je vous conseille de vous servir de l'Auto-Connect. Une fois qu'on y a goûté on ne peut plus s'en passer. 😊

TP : zNavigo, le navigateur web des Zéros !

Depuis le temps que vous pratiquez Qt, vous avez acquis sans vraiment le savoir les capacités de base pour réaliser des programmes complexes. Le but d'un TP comme celui-ci, c'est de vous montrer justement que vous êtes capables de mener à bien des projets qui auraient pu vous sembler complètement fous il y a quelques temps.

Vous ne rêvez pas : le but de ce TP sera de... **réaliser un navigateur web** !



Quoi ? Je suis capable de faire ça moi ? 🤔

Oui, et nous allons voir comment dans ce chapitre !

Nous allons commencer dans un premier temps par découvrir la notion de *moteur web*, pour bien comprendre comment fonctionnent les autres navigateurs. Puis, nous mettrons en place le plan du développement de notre programme afin de nous assurer que nous partons dans la bonne direction et que nous n'oublions rien.

Firefox n'a qu'à bien se tenir. 🐘

Les navigateurs et les moteurs web

Comme toujours, il faut d'abord prendre le temps de *réfléchir à son programme* avant de foncer le coder tête baissée. C'est ce qu'on appelle la phase de **conception**.

Je sais, je me répète à chaque fois, mais c'est vraiment parce que c'est très important. Si je vous dis "*faites-moi un navigateur web*" et que vous créez de suite un nouveau projet en vous demandant ce que vous allez bien pouvoir mettre dans le *main*, ben... c'est le ramassage assuré. 🤡

Pour moi, la conception est l'étape la plus difficile du projet. Plus difficile même que le code. En effet, si vous concevez bien votre programme, si vous réfléchissez bien à la façon dont il doit fonctionner, vous aurez simplifié à l'avance votre projet et vous n'aurez pas à écrire des lignes de code difficiles inutilement.

Dans un premier temps, je vais vous expliquer comment fonctionne un navigateur web. Un peu de culture générale à ce sujet vous permettra de mieux comprendre ce que vous avez à faire (et ce que vous n'avez pas à faire).

Je vous donnerai ensuite quelques conseils pour organiser votre code : quelles classes créer, par quoi commencer, etc.

Les principaux navigateurs

Commençons par le commencement : vous savez ce qu'est un navigateur web ?

Bon, je ne me moque pas de vous, mais il vaut mieux être sûr de ne perdre personne. 🤪

Un navigateur web est un programme qui permet de consulter des sites web.

Parmi les plus connus d'entre eux, citons Internet Explorer, Mozilla Firefox ou encore Safari. Mais il y en a aussi beaucoup

d'autres, certes moins utilisés, comme Opera, Konqueror, Epiphany, Maxthon, Lynx..

Je vous rassure, il n'est pas nécessaire de tous les connaître pour pouvoir prétendre en créer un.

Par contre, ce qu'il faut que vous sachiez, c'est que chacun de ces navigateurs est constitué de ce qu'on appelle un **moteur web**. Qu'est-ce que c'est que cette bête-là ?

Le moteur web

Tous les sites web sont écrits en langage HTML (ou XHTML). Voici un exemple de code HTML permettant de créer une page très simple :

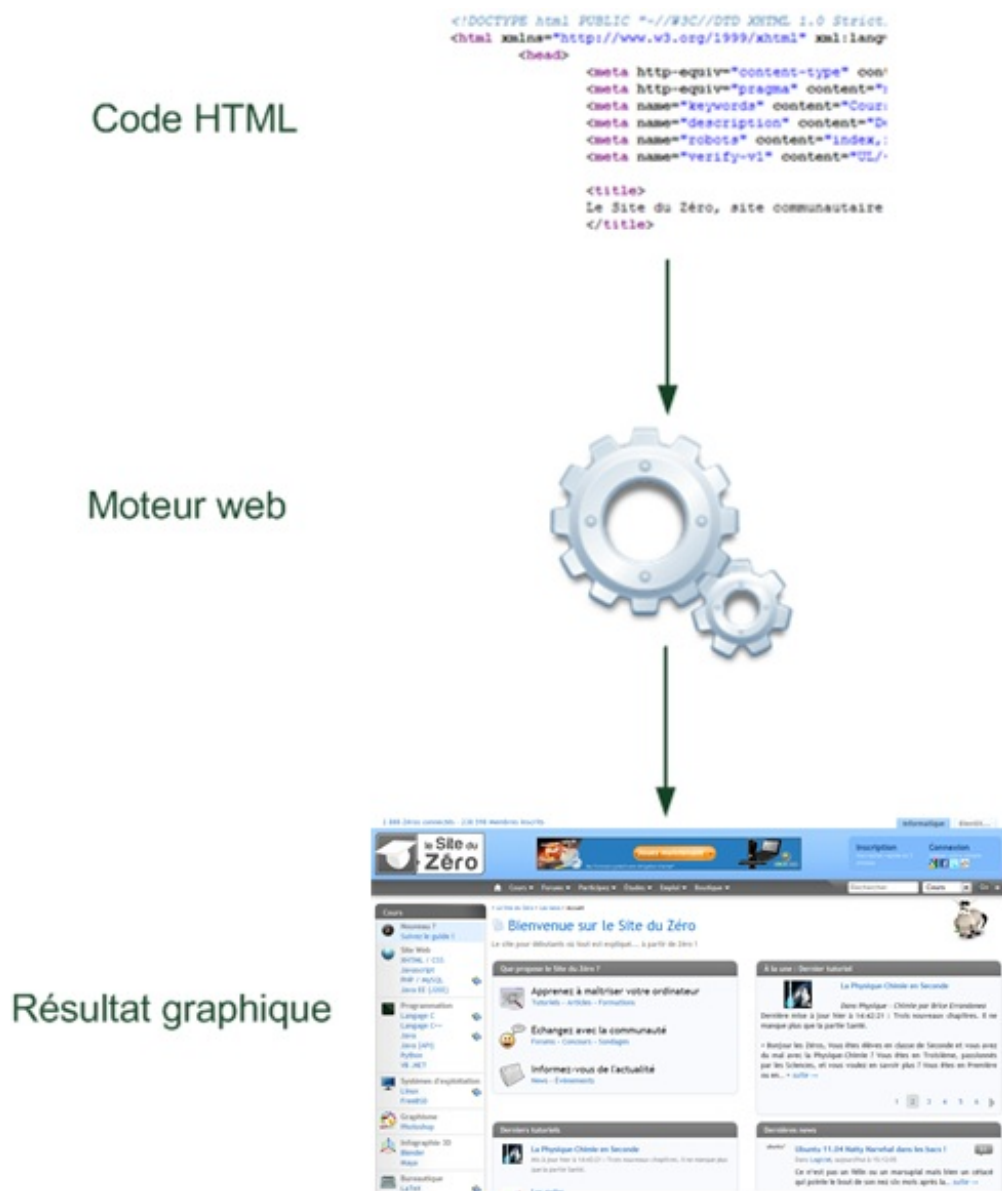
Code : HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
  <head>
    <title>Bienvenue sur mon site !</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  </head>
  <body>
  </body>
</html>
```

C'est bien joli tout ce code, mais ça ne ressemble pas au résultat visuel qu'on a l'habitude de voir lorsqu'on navigue sur le web.

L'objectif est justement de transformer ce code en un résultat visuel : le site web. C'est le rôle du moteur web.

Voici son fonctionnement, résumé dans un schéma très simple :



Ca n'a l'air de rien, mais c'est un travail difficile : réaliser un moteur web est très délicat. C'est généralement le fruit des efforts de nombreux programmeurs experts (et encore, ils avouent avoir du mal 🤔). Certains moteurs sont meilleurs que d'autres, mais aucun n'est parfait ni complet. Comme le web est en perpétuelle évolution, il est peu probable qu'un moteur parfait sorte un jour.

Quand on programme un navigateur, on utilise généralement le moteur web sous forme de bibliothèque. Le moteur web n'est donc pas un programme, mais il est utilisé par des programmes.



Ce sera plus clair avec un schéma. 🤔
Regardons comment est constitué Firefox par exemple :



On voit que le navigateur (en vert) "contient" le moteur web (en jaune au centre).



La partie en vert est habituellement appelée le "chrome", pour désigner l'interface.



Mais c'est nul ! Alors le navigateur web c'est juste les 2-3 boutons en haut et c'est tout ?

Oh non ! Loin de là.

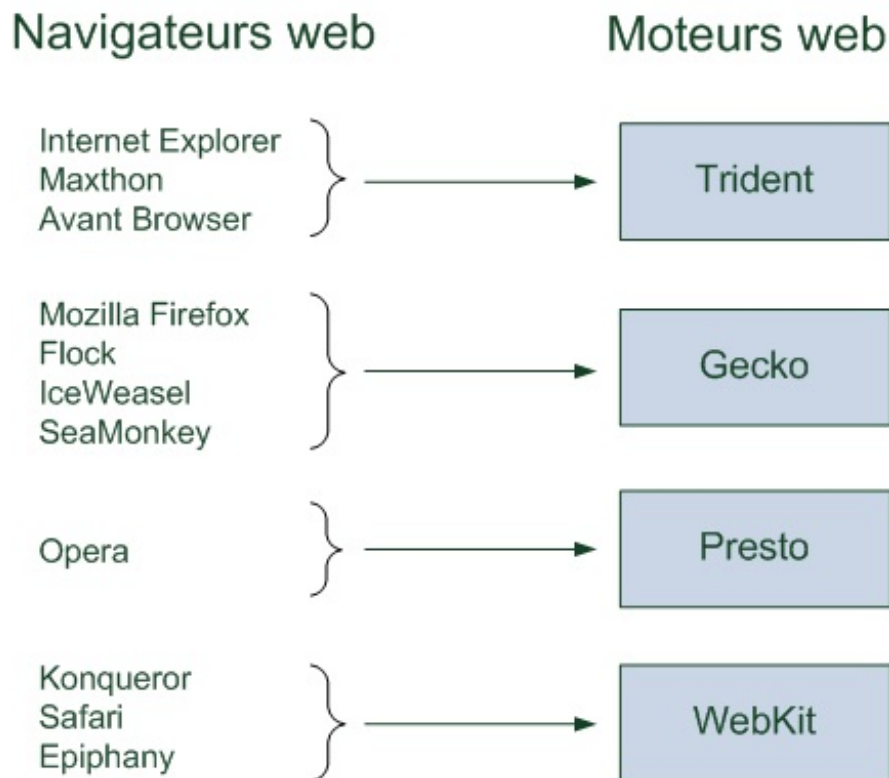
Le navigateur ne se contente pas de gérer les boutons "Précédente", "Suivante", "Actualiser", etc. C'est aussi lui qui gère les marque-pages (favoris), le système d'onglets, les options d'affichage, la barre de recherche, etc.

Tout cela représente déjà un énorme travail ! En fait, les développeurs de Firefox ne sont pas les mêmes que ceux qui développent son moteur web. Il y a des équipes séparées, tellement chacun de ces éléments représente du travail.

Les principaux navigateurs et leurs moteurs

Un grand nombre de navigateurs ne s'occupent pas du moteur web. Ils en utilisent un "tout prêt".

De nombreux navigateurs sont basés sur le même moteur web. Voici un petit schéma de mon crû qui vous permet de vous donner une idée un peu de "qui utilise quoi" :



Les noms des moteurs web ne sont pas connus du grand public. D'ailleurs, il est probable que vous n'ayez entendu parler d'aucun d'eux jusqu'à aujourd'hui. Ce qui est connu, c'est le navigateur, alors que c'est le moteur web qui se tape tout le sale boulot. 🤖

Je n'ai pas mis tous les navigateurs et moteurs web existants, mais cela permet déjà d'avoir une bonne idée de ce qui se passe. Comme vous le voyez, rares sont les navigateurs à avoir leur propre moteur web. On peut noter l'exception d'Opera (et encore, le moteur a été revendu à Adobe qui ne voulait pas en coder un pour son logiciel Dreamweaver).

Tout ça pour dire quoi ? Eh bien déjà que **créer un moteur web n'est ni de votre niveau, ni du mien**. Comme de nombreux navigateurs, nous en utiliserons un déjà existant.

Lequel ? Eh bien il se trouve que Qt (oui, parce qu'on parle de Qt ici, j'espère que vous n'avez pas oublié 😊), Qt donc vous propose depuis peu d'utiliser le moteur WebKit dans vos programmes. C'est donc ce moteur-là que nous allons utiliser pour créer notre navigateur.

Configurer son projet pour utiliser WebKit

WebKit est un des nombreux modules de Qt. Il ne fait pas partie du module "GUI", dédié à la création de fenêtres, il s'agit d'un module à part.

Pour pouvoir l'utiliser, il faudra modifier le fichier .pro du projet pour que Qt sache qu'il a besoin de charger WebKit. Voici un exemple de fichier .pro qui indique que le projet utilise WebKit :

Code : Autre

```
#####
# Automatically generated by qmake (2.01a) mer. 18. juin 11:49:49 2008
#####

TEMPLATE = app
QT += webkit
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
```

```
# Input
HEADERS += FenPrincipale.h
SOURCES += FenPrincipale.cpp main.cpp
```

D'autre part, vous devrez rajouter l'include suivant dans les fichiers de votre code source faisant appel à WebKit :

Code : C++

```
#include <QtWebKit>
```

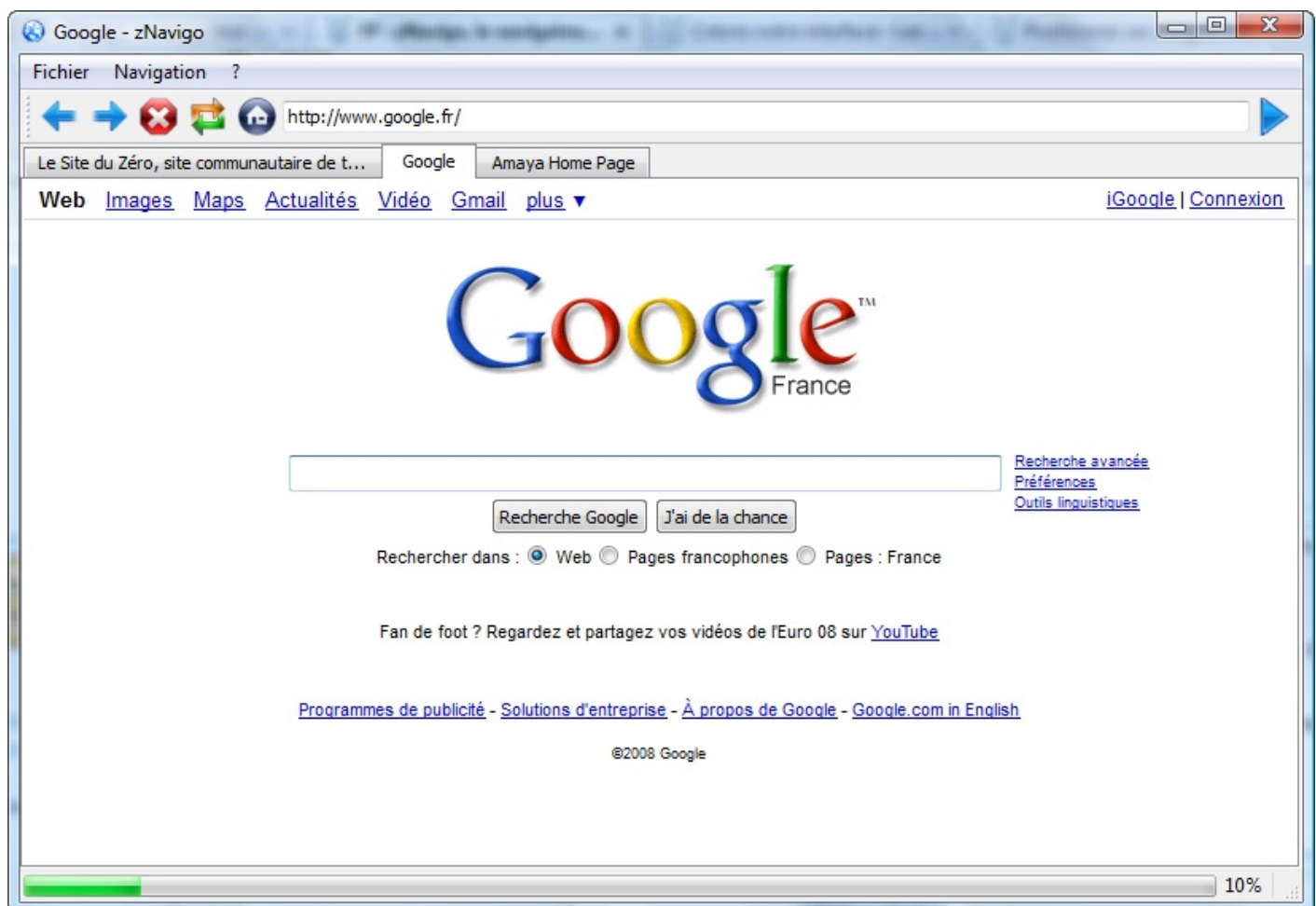
Enfin, il faudra joindre 2 nouvelles DLL à votre programme pour qu'il fonctionne : QtWebKit4.dll et QtNetwork4.dll.

Ouf, tout est prêt. 😊

Organisation du projet

Objectif

Avant d'aller plus loin, il me semble indispensable de vous montrer à quoi doit ressembler le navigateur une fois terminé. Votre objectif est de réaliser un navigateur web semblable à celui-ci :



Parmi les fonctionnalités de ce super navigateur, affectueusement nommé "zNavigo", on compte :

- Accès aux pages précédentes et suivantes
- Arrêter le chargement de la page
- Actualiser la page

- Retour à la page d'accueil
- Saisie d'une adresse
- Navigation par onglets
- Affichage du pourcentage de chargement dans la barre d'état

Le menu "Fichier" propose d'ouvrir et de fermer un onglet, ainsi que de quitter le programme.

Le menu "Navigation" reprend le contenu de la barre d'outils (ce qui est très facile à faire grâce aux QAction je vous le rappelle).

Le menu "?" (aide) propose d'afficher les fenêtres "A propos..." et "A propos de Qt..." qui donnent des informations respectivement sur notre programme et sur Qt.

Ca n'a l'air de rien comme ça, mais ça représente déjà un sacré boulot !

Si vous avez du mal dans un premier temps, vous pouvez vous épargner la gestion des onglets... mais moi j'ai trouvé que c'était un peu trop simple sans les onglets, alors j'ai choisi de vous faire jouer avec, histoire de corser le tout. 🤪

Les fichiers du projet

J'ai l'habitude de faire une classe par fenêtre. Comme notre projet ne sera constitué (au moins dans un premier temps) que d'une seule fenêtre, nous aurons donc les fichiers suivants :

- main.cpp
- FenPrincipale.h
- FenPrincipale.cpp

Si vous voulez utiliser les mêmes icônes que moi, les voici :



Notez que la dernière est l'icône du programme (affichée en haut à gauche).



Toutes ces icônes sont sous licence LGPL et proviennent du site <http://www.everaldo.com>

Utiliser QWebView pour afficher une page web

Le **QWebView** est le principal nouveau widget que vous aurez besoin d'utiliser dans ce chapitre. Il permet d'afficher une page web. C'est lui le **moteur web**.

Vous ne savez pas vous en servir, mais vous savez maintenant [lire la doc](#). Vous allez voir, ce n'est pas bien difficile !

Regardez en particulier les signaux et slots proposés par le QWebView. Il y a tout ce qu'il faut savoir pour, par exemple, connaître le pourcentage de chargement de la page pour le répercuter sur la barre de progression de la barre d'état (signal

`loadProgress(int)`). 😊

Comme l'indique la doc, pour créer le widget et charger une page, c'est très simple :

Code : C++

```
QWebView *pageWeb = new QWebView;
pageWeb->load(QUrl("http://www.siteduzero.com/"));
```

Voilà c'est tout ce que je vous expliquerai sur QWebView, pour le reste lisez la doc. 😊

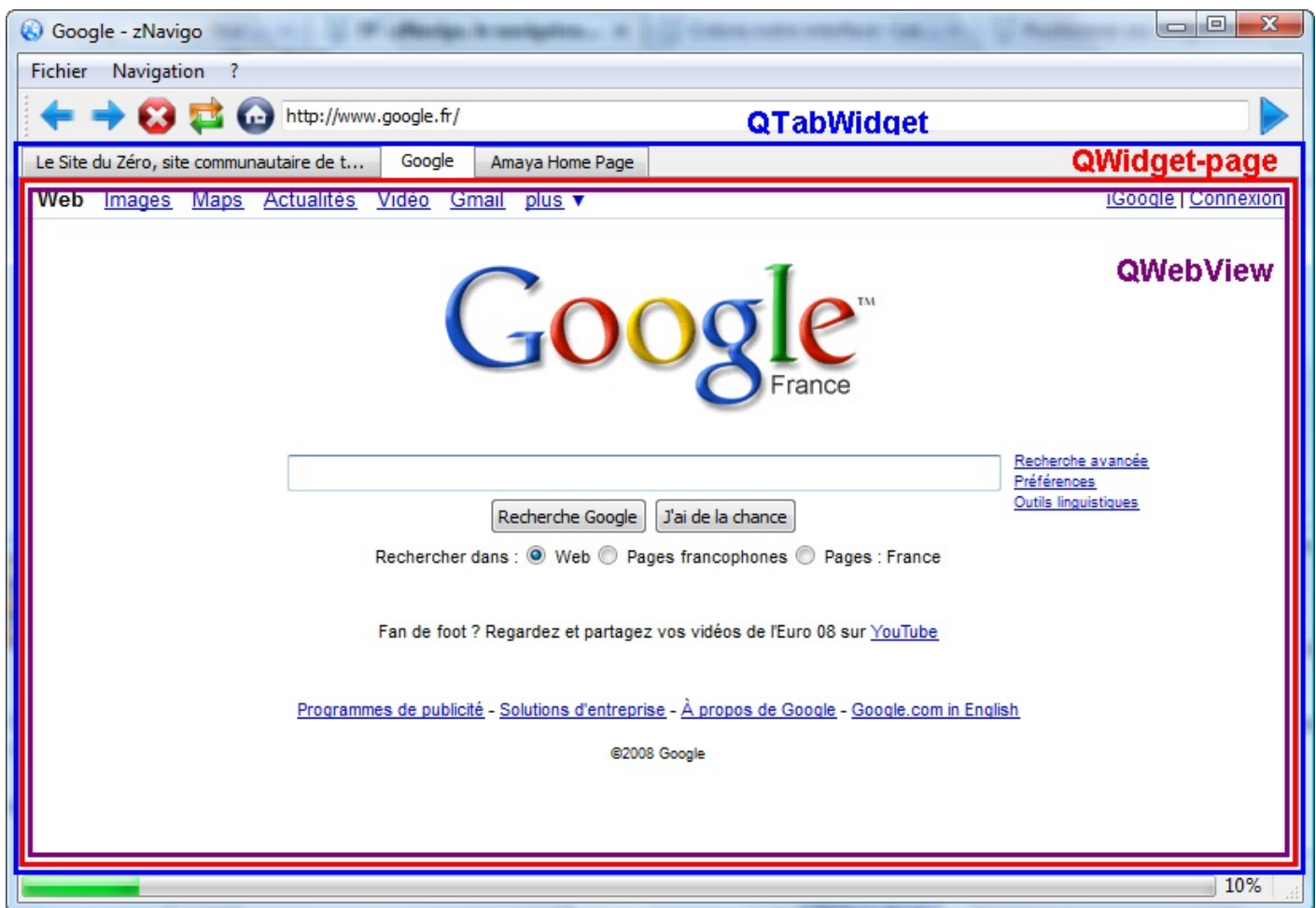
La navigation par onglets

Le problème de QWebView, c'est qu'il ne permet d'afficher qu'une seule page web à la fois. Il ne gère pas la navigation par onglets. Il va falloir implémenter le système d'onglets nous-mêmes.

Vous n'avez jamais entendu parler de QTabWidget ? Si si, souvenez-vous, nous l'avons découvert dans un des chapitres précédents. Ce widget-conteneur est capable d'accueillir n'importe quels widgets... comme un QWebView ! En combinant un QTabWidget et des QWebView (un par onglet), vous pourrez reconstituer un véritable navigateur par onglets !

Une petite astuce toutefois, qui pourra vous être bien utile : savoir retrouver un widget contenu dans un widget parent. Comme vous le savez, le QTabWidget utilise des sous-widgets pour gérer chacune des pages. Ces sous-widgets sont généralement des QWidget génériques (invisibles), qui servent à contenir d'autres widgets.

Dans notre cas : **QTabWidget** contient des **QWidget** (pages d'onglet) qui eux-mêmes contiennent chacun un **QWebView** (la page web).



La méthode findChild (définie dans QObject) permet de retrouver le widget enfant contenu dans le widget parent.

Par exemple, si je connais le QWidget "pageOnglet", je peux retrouver le QWebView qu'il contient comme ceci :

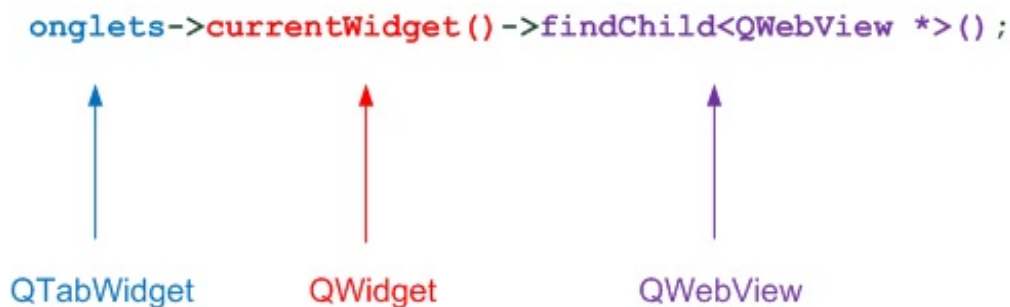
Code : C++

```
QWebView *pageWeb = pageOnglet->findChild<QWebView *>();
```


Mieux encore, je vous donne la méthode toute faite qui permet de retrouver le QWebView actuellement visualisé par l'utilisateur :

Code : C++

```
QWebView *FenPrincipale::pageActuelle()
{
    return onglets->currentWidget()->findChild<QWebView *>();
}
```



"onglets" correspond au QTabWidget.

Sa méthode currentWidget() permet d'obtenir un pointeur vers le QWidget qui sert de page pour la page actuellement affichée. On demande ensuite à retrouver le QWebView que le QWidget contient à l'aide de la méthode findChild(). Cette méthode utilise les templates C++ (avec <QWebView *>). Je ne vais pas rentrer dans les détails ici car ce serait un peu trop long, mais en gros cela permet de faire en sorte que la méthode retourne bien un QWebView* (sinon elle n'aurait pas su quoi renvoyer).

J'admets, c'est un petit peu compliqué, mais au moins ça pourra vous aider. 😊

Let's go !

Voilà, vous savez déjà tout ce qu'il faut pour vous en sortir.

Notez que ce TP fait la part belle à la QMainWindow, n'hésitez donc pas à relire ce chapitre dans un premier temps pour bien vous remémorer son fonctionnement.

Pour ma part, j'ai choisi de coder la fenêtre "à la main" (pas de Qt Designer donc) car celle-ci est un peu complexe.

Comme il y a beaucoup d'initialisations à faire dans le constructeur, je vous conseille de les placer dans des méthodes que vous appellerez depuis le constructeur pour améliorer la lisibilité :

Code : C++

```
FenPrincipale::FenPrincipale()
{
    creerActions();
    creerMenus();
    creerBarresOutils();

    /* Autres initialisations */
}
```

Bon courage ! 😊

Génération de la fenêtre principale

Commençons par les choses simples (et un peu répétitives).

main.cpp

Tout d'abord le main.cpp, qui ne devrait pas vous perturber :

Code : C++

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include "FenPrincipale.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    // Traduction des chaînes prédéfinies par Qt dans notre langue
    QString locale = QLocale::system().name();
    QTranslator translator;
    translator.load(QString("qt_") + locale,
        QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    // Ouverture de la fenêtre principale du navigateur
    FenPrincipale principale;
    principale.show();

    return app.exec();
}
```

Je me contente d'ouvrir la fenêtre principale. Les quelques lignes de code au début sont celles que je vous avais données il y a quelques chapitres, pour faire en sorte que les chaînes de caractères de base (Yes / No) soient traduites en français dans l'application.

Maintenant, créons la classe FenPrincipale, notre plus gros morceau.

FenPrincipale.h (première version)

Dans un premier temps, je ne crée que le squelette de la classe et ses premières méthodes, j'en rajouterai d'autres au fur et à mesure si besoin est.

Code : C++

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>
#include <QtWebKit>

class FenPrincipale : public QMainWindow
{
    Q_OBJECT

public:
    FenPrincipale();

private:

```



```

        void creerActions();
        void creerMenus();
        void creerBarresOutils();
        void creerBarreEtat();

    private slots:

    private:
        QTabWidget *onglets;

        QAction *actionNouvelOnglet;
        QAction *actionFermerOnglet;
        QAction *actionQuitter;
        QAction *actionAPropos;
        QAction *actionAProposQt;
        QAction *actionPrecedente;
        QAction *actionSuivante;
        QAction *actionStop;
        QAction *actionActualiser;
        QAction *actionAccueil;
        QAction *actionGo;

        QLineEdit *champAdresse;
        QProgressBar *progression;
};

#endif

```

La classe hérite de QMainWindow comme prévu. J'ai inclus QtGui et QtWebKit pour pouvoir utiliser le module GUI et le module WebKit (moteur web).

Mon idée c'est, comme je vous l'avais dit, de couper le constructeur en plusieurs sous-méthodes qui s'occupent chacune de créer une section différente de la QMainWindow : actions, menus, barre d'outils, barre d'état...

J'ai prévu une section pour les slots personnalisés mais je n'ai encore rien mis, je verrai au fur et à mesure.

Enfin, j'ai préparé les principaux attributs de la classe. En fin de compte, à part de nombreuses QAction, il n'y en a pas beaucoup. Je n'ai même pas eu besoin de mettre des objets de type QWebView : ceux-ci seront créés à la volée au cours du programme et on pourra les retrouver grâce à la méthode pageActuelle() que je vous ai donnée un peu plus tôt.

Voyons voir l'implémentation du constructeur et de ses sous-méthodes qui génèrent le contenu de la fenêtre.

Construction de la fenêtre

Direction FenPrincipale.cpp, on commence par le constructeur :

Code : C++

```

#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Génération des widgets de la fenêtre principale
    creerActions();
    creerMenus();
    creerBarresOutils();
    creerBarreEtat();

    // Génération des onglets et chargement de la page d'accueil
    onglets = new QTabWidget;
    onglets->addTab(creerOngletPageWeb(tr("http://www.siteduzero.com")),

```

```

tr(" (Nouvelle page) "));
connect(onglets, SIGNAL(currentChanged(int)), this,
SLOT(changementOnglet(int)));
setCentralWidget(onglets);

// Définition de quelques propriétés de la fenêtre
setMinimumSize(500, 350);
setWindowIcon(QIcon("images/znavigo.png"));
setWindowTitle(tr("zNavigo"));
}

```

Nous allons voir juste après le code des méthodes `creerActions()`, `creerMenus()`, etc. Ce code est un peu long et répétitif, pas très intéressant mais il fallait le faire.

Par contre, ce qui est intéressant ensuite dans le constructeur, c'est que l'on crée le `QTabWidget` et on lui ajoute un premier onglet. Pour la création d'un onglet, on va faire appel à une méthode "maison" `creerOngletPageWeb()` qui va se charger de créer le `QWidget`-page de l'onglet, ainsi que de créer un `QWebView` et de lui faire charger la page web envoyée en paramètre ("<http://www.siteduzero.com>" sera donc la page d'accueil par défaut 🤖).

Vous noterez que l'on utilise la fonction de `tr()` partout, au cas où on voudrait traduire le programme par la suite. C'est une bonne habitude à prendre, même si on n'a pas forcément l'intention de traduire le programme au début (on peut toujours changer d'avis après).

On connecte enfin et surtout le signal `currentChanged()` du `QTabWidget` à un slot personnalisé `changementOnglet()` que l'on va devoir écrire. Ce slot sera appelé à chaque fois que l'utilisateur change d'onglet, pour, par exemple, mettre à jour l'URL dans la barre d'adresse ainsi que le titre de la page affiché en haut de la fenêtre.

Bon, il faut maintenant écrire les méthodes de génération des actions, des menus, etc. C'était un peu long et fastidieux mais je suis arrivé jusqu'au bout. 🤖

Ne vous laissez pas impressionner par la taille du code, je n'ai pas tout écrit d'un coup, j'y suis allé petit à petit.

Code : C++

```

void FenPrincipale::creerActions()
{
    actionNouvelOnglet = new QAction(tr("&Nouvel onglet"), this);
    actionNouvelOnglet->setShortcut(tr("Ctrl+T"));
    connect(actionNouvelOnglet, SIGNAL(triggered()), this,
SLOT(nouvelOnglet()));
    actionFermerOnglet = new QAction(tr("&Fermer l'onglet"), this);
    actionFermerOnglet->setShortcut(tr("Ctrl+W"));
    connect(actionFermerOnglet, SIGNAL(triggered()), this,
SLOT(fermerOnglet()));
    actionQuitter = new QAction(tr("&Quitter"), this);
    actionQuitter->setShortcut(tr("Ctrl+Q"));
    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));

    actionPrecedente = new QAction(QIcon("images/precedente.png"),
tr("&Precedente"), this);
    actionPrecedente->setShortcut(QKeySequence::Back);
    connect(actionPrecedente, SIGNAL(triggered()), this,
SLOT(precedente()));
    actionSuivante = new QAction(QIcon("images/suivante.png"),
tr("&Suivante"), this);
    actionSuivante->setShortcut(QKeySequence::Forward);
    connect(actionSuivante, SIGNAL(triggered()), this,
SLOT(suivante()));
    actionStop = new QAction(QIcon("images/stop.png"), tr("S&top"),
this);
    connect(actionStop, SIGNAL(triggered()), this, SLOT(stop()));
    actionActualiser = new QAction(QIcon("images/actualiser.png"),
tr("&Actualiser"), this);
    actionActualiser->setShortcut(QKeySequence::Refresh);
}

```

```

        connect(actionActualiser, SIGNAL(triggered()), this,
        SLOT(actualiser()));
        actionAccueil = new QAction(QIcon("images/accueil.png"),
        tr("A&ccueil"), this);
        connect(actionAccueil, SIGNAL(triggered()), this,
        SLOT(accueil()));
        actionGo = new QAction(QIcon("images/go.png"), tr("A&ller à"),
        this);
        connect(actionGo, SIGNAL(triggered()), this,
        SLOT(chargerPage()));

        actionAPropos = new QAction(tr("&A propos..."), this);
        connect(actionAPropos, SIGNAL(triggered()), this,
        SLOT(aPropos()));
        actionAPropos->setShortcut(QKeySequence::HelpContents);
        actionAProposQt = new QAction(tr("A propos de &Qt..."), this);
        connect(actionAProposQt, SIGNAL(triggered()), qApp,
        SLOT(aboutQt()));
    }

    void FenPrincipale::creerMenus()
    {
        QMenu *menuFichier = menuBar()->addMenu(tr("&Fichier"));

        menuFichier->addAction(actionNouvelOnglet);
        menuFichier->addAction(actionFermerOnglet);
        menuFichier->addSeparator();
        menuFichier->addAction(actionQuitter);

        QMenu *menuNavigation = menuBar()->addMenu(tr("&Navigation"));

        menuNavigation->addAction(actionPrecedente);
        menuNavigation->addAction(actionSuivante);
        menuNavigation->addAction(actionStop);
        menuNavigation->addAction(actionActualiser);
        menuNavigation->addAction(actionAccueil);

        QMenu *menuAide = menuBar()->addMenu(tr("&?"));

        menuAide->addAction(actionAPropos);
        menuAide->addAction(actionAProposQt);
    }

    void FenPrincipale::creerBarresOutils()
    {
        champAdresse = new QLineEdit;
        connect(champAdresse, SIGNAL(returnPressed()), this,
        SLOT(chargerPage()));

        QToolBar *toolBarNavigation = addToolBar(tr("Navigation"));

        toolBarNavigation->addAction(actionPrecedente);
        toolBarNavigation->addAction(actionSuivante);
        toolBarNavigation->addAction(actionStop);
        toolBarNavigation->addAction(actionActualiser);
        toolBarNavigation->addAction(actionAccueil);
        toolBarNavigation->addWidget(champAdresse);
        toolBarNavigation->addAction(actionGo);
    }

    void FenPrincipale::creerBarreEtat()
    {
        progression = new QProgressBar(this);
        progression->setVisible(false);
        progression->setMaximumHeight(14);
        statusBar()->addWidget(progression, 1);
    }

```

Ce code ne fait rien d'extraordinairement nouveau, je ne vois pas trop ce que je pourrais commenter. Il y a beaucoup de connexions à des slots personnalisés que l'on devra écrire.

C'était la partie "longue" du code, mais certainement pas la plus complexe.

Voyons maintenant quelques méthodes qui s'occupent de gérer les onglets...

Méthodes de gestion des onglets

En fait, il n'y a que 2 méthodes dans cette catégorie :

- **creerOngletPageWeb()** : je vous en ai parlé dans le constructeur, elle se charge de créer un QWidget-page ainsi qu'un QWebView à l'intérieur, et de retourner ce QWidget-page à l'appelant pour qu'il puisse créer le nouvel onglet.
- **pageActuelle()** : une méthode bien pratique que je vous ai donnée un peu plus tôt, qui permet à tout moment d'obtenir un pointeur vers le QWebView de l'onglet actuellement sélectionné.

Voici ces méthodes :

Code : C++

```
QWidget *FenPrincipale::creerOngletPageWeb(QString url)
{
    QWidget *pageOnglet = new QWidget;
    QWebView *pageWeb = new QWebView;

    QVBoxLayout *layout = new QVBoxLayout;
    layout->setContentsMargins(0,0,0,0);
    layout->addWidget(pageWeb);
    pageOnglet->setLayout(layout);

    if (url.isEmpty())
    {
        pageWeb->load(QUrl(tr("html/page_blanche.html")));
    }
    else
    {
        if (url.left(7) != "http://")
        {
            url = "http://" + url;
        }
        pageWeb->load(QUrl(url));
    }

    // Gestion des signaux envoyés par la page web
    connect(pageWeb, SIGNAL(titleChanged(QString)), this,
    SLOT(changementTitre(QString)));
    connect(pageWeb, SIGNAL(urlChanged(QUrl)), this,
    SLOT(changementUrl(QUrl)));
    connect(pageWeb, SIGNAL(loadStarted()), this,
    SLOT(changementDebut()));
    connect(pageWeb, SIGNAL(loadProgress(int)), this,
    SLOT(changementEnCours(int)));
    connect(pageWeb, SIGNAL(loadFinished(bool)), this,
    SLOT(changementTermine(bool)));

    return pageOnglet;
}

QWebView *FenPrincipale::pageActuelle()
```

```
{
    return onglets->currentWidget()->findChild<QWebView *>();
}
```

Je ne commente pas `pageActuelle()`, je l'ai déjà fait auparavant.

Pour ce qui est de `creerOngletPageWeb()`, elle crée comme prévu un `QWidget` et elle place un nouveau `QWebView` à l'intérieur. La page web charge l'URL indiquée en paramètre, et rajoute le "http://" en préfixe si celui-ci a été oublié.

Si aucune URL n'a été spécifiée, on charge une page blanche. J'ai pour l'occasion créé un fichier HTML vide, placé dans un sous-dossier "html" du programme.

On connecte plusieurs signaux intéressants envoyés par le `QWebView`, qui, à mon avis, parlent d'eux-mêmes : "Le titre a changé", "L'URL a changé", "Début du chargement", "Chargement en cours", "Chargement terminé".

Bref, rien de sorcier, mais ça fait encore tout plein de slots personnalisés à écrire tout ça ! 🤖

Les slots personnalisés

Bon, il y a de quoi faire. Reprenons notre `FenPrincipale.h`, que voici maintenant en version complète avec toutes les méthodes et tous les slots.

FenPrincipale.h (version complète)

Code : C++

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>
#include <QtWebKit>

class FenPrincipale : public QMainWindow
{
    Q_OBJECT

public:
    FenPrincipale();

private:
    void creerActions();
    void creerMenus();
    void creerBarresOutils();
    void creerBarreEtat();
    QWidget *creerOngletPageWeb(QString url = "");
    QWebView *pageActuelle();

private slots:
    void precedente();
    void suivante();
    void accueil();
    void stop();
    void actualiser();

    void aPropos();
    void nouvelOnglet();
    void fermerOnglet();
    void chargerPage();
    void changementOnglet(int index);

    void changementTitre(const QString & titreCompleet);
    void changementUrl(const QUrl & url);
    void chargementDebut();
    void chargementEnCours(int pourcentage);
    void chargementTermine(bool ok);
}
```

```
private:
    QTabWidget *onglets;

    QAction *actionNouvelOnglet;
    QAction *actionFermerOnglet;
    QAction *actionQuitter;
    QAction *actionAPropos;
    QAction *actionAProposQt;
    QAction *actionPrecedente;
    QAction *actionSuivante;
    QAction *actionStop;
    QAction *actionActualiser;
    QAction *actionAccueil;
    QAction *actionGo;

    QLineEdit *champAdresse;
    QProgressBar *progression;
};

#endif
```

Les lignes ajoutées par rapport à la fois précédente ont été surlignées.

Maintenant implémentons tout ça. 😊

Implémentation des slots

Slots appelés par les actions de la barre d'outils

Commençons par les actions de la barre d'outils :

Code : C++

```
void FenPrincipale::precedente()
{
    pageActuelle()->back();
}

void FenPrincipale::suivante()
{
    pageActuelle()->forward();
}

void FenPrincipale::accueil()
{
    pageActuelle()->load(QUrl(tr("http://www.siteduzero.com")));
}

void FenPrincipale::stop()
{
    pageActuelle()->stop();
}

void FenPrincipale::actualiser()
{
    pageActuelle()->reload();
}
```

On utilise la (très) pratique fonction `pageActuelle()` pour obtenir un pointeur vers le `QWebView` que l'utilisateur est en train de regarder (histoire d'affecter la page web de l'onglet en cours, et pas les autres).

Toutes ces méthodes, comme `back()` et `forward()`, sont des slots. On les appelle ici comme si c'étaient de simples méthodes.



Pourquoi ne pas avoir connecté directement les signaux envoyés par les `QAction` aux slots du `QWebView` ?

On aurait pu s'il n'y avait pas eu d'onglets. Le problème justement ici, c'est qu'on gère plusieurs onglets différents.

Par exemple, on ne pouvait pas connecter lors de sa création la `QAction` "actualiser" au `QWebView`... parce que le `QWebView` à actualiser dépend de l'onglet actuellement sélectionné !

Voilà donc pourquoi on passe par un petit slot maison qui va d'abord chercher à savoir quel est le `QWebView` que l'on est en train de visualiser pour être sûr qu'on recharge la bonne page. 😊

Slots appelés par d'autres actions des menus

Voici les slots appelés par les actions des menus suivants :

- Nouvel onglet
- Fermer l'onglet
- A propos...

Code : C++

```
void FenPrincipale::aPropos()
{
    QMessageBox::information(this, tr("A propos..."), tr("zNavigo
est un projet réalisé pour illustrer les tutoriels C++ du <a
href=\"http://www.siteduzero.com\">Site du Zéro</a>.<br />Les images
de ce programme ont été créées par <a
href=\"http://www.veraldo.com\">Everaldo Coelho</a>"));
}

void FenPrincipale::nouvelOnglet()
{
    int indexNouvelOnglet = onglets->addTab(creerOngletPageWeb(),
tr("(Nouvelle page)"));
    onglets->setCurrentIndex(indexNouvelOnglet);

    champAdresse->setText("");
    champAdresse->setFocus(Qt::OtherFocusReason);
}

void FenPrincipale::fermerOnglet()
{
    // On ne doit pas fermer le dernier onglet, sinon le QTabWidget
ne marche plus
    if (onglets->count() > 1)
    {
        onglets->removeTab(onglets->currentIndex());
    }
    else
    {
        QMessageBox::critical(this, tr("Erreur"), tr("Il faut au
moins un onglet !"));
    }
}
```

Le slot `aPropos()` se contente d'afficher une boîte de dialogue.

`nouvelOnglet()` rajoute un nouvel onglet à l'aide de la méthode `addTab()` du `QTabWidget`, comme on l'avait fait dans le constructeur. Pour que le nouvel onglet s'affiche immédiatement, on force son affichage avec `setCurrentIndex()` qui se sert de l'index (numéro) de l'onglet que l'on vient de créer.

On vide la barre d'adresse et on lui donne le focus, c'est-à-dire que le curseur est directement placé dedans pour que l'utilisateur puisse écrire une URL.



L'action "Nouvel onglet" a comme raccourci "Ctrl+T", ce qui permet d'ouvrir un onglet à tout moment à l'aide du raccourci clavier correspondant.

Vous pouvez aussi ajouter un bouton dans la barre d'outils pour ouvrir un nouvel onglet ou, encore mieux, rajouter un mini-bouton dans un des coins du `QTabWidget`. Regardez du côté de la méthode `setCornerWidget()`.

`fermerOnglet()` supprime l'onglet actuellement sélectionné. Il vérifie au préalable que l'on n'est pas en train d'essayer de supprimer le dernier onglet, auquel cas le `QTabWidget` n'aurait plus lieu d'exister. Un système à onglets sans onglets, ça fait désordre. 😊

Slots de chargement d'une page et de changement d'onglet

Ces slots sont appelés respectivement lorsqu'on demande à charger une page (appui sur la touche Entrée après avoir écrit une URL, ou clic sur le bouton tout à droite de la barre d'outils) et lorsqu'on change d'onglet.

Code : C++

```
void FenPrincipale::chargerPage()
{
    QString url = champAdresse->text();

    // On rajoute le "http://" s'il n'est pas déjà dans l'adresse
    if (url.left(7) != "http://")
    {
        url = "http://" + url;
        champAdresse->setText(url);
    }

    pageActuelle()->load(QUrl(url));
}

void FenPrincipale::changementOnglet(int index)
{
    changementTitre(pageActuelle()->title());
    changementUrl(pageActuelle()->url());
}
```

On vérifie au préalable que l'utilisateur a mis le préfixe `http://`, et si ce n'est pas le cas on le rajoute (sinon l'adresse n'est pas valide).

Lorsque l'utilisateur change d'onglet, on met à jour 2 choses sur la fenêtre : le titre de la page, affiché tout en haut de la fenêtre et sur un onglet, et l'URL inscrite dans la barre d'adresse.

`changementTitre()` et `changementUrl()` sont des slots personnalisés, que l'on se permet d'appeler comme n'importe quelle méthode. Ces slots sont aussi automatiquement appelés lorsque le `QWebView` envoie les signaux correspondants.

Voyons voir comment implémenter ces slots...

Slots appelés lorsqu'un signal est envoyé par le QWebView

Lorsque le `QWebView` s'active, il va envoyer des signaux. Ceux-ci sont connectés à des slots personnalisés de notre fenêtre. Les

voici :

Code : C++

```
void FenPrincipale::changementTitre(const QString & titreComplet)
{
    QString titreCourt = titreComplet;

    // On tronque le titre pour éviter des onglets trop larges
    if (titreComplet.size() > 40)
    {
        titreCourt = titreComplet.left(40) + "...";
    }

    setWindowTitle(titreCourt + " - " + tr("zNavigo"));
    onglets->setTabText(onglets->currentIndex(), titreCourt);
}

void FenPrincipale::changementUrl(const QUrl & url)
{
    if (url.toString() != tr("html/page_blanche.html"))
    {
        champAdresse->setText(url.toString());
    }
}

void FenPrincipale::chargementDebut()
{
    progression->setVisible(true);
}

void FenPrincipale::chargementEnCours(int pourcentage)
{
    progression->setValue(pourcentage);
}

void FenPrincipale::chargementTermine(bool ok)
{
    progression->setVisible(false);
    statusBar()->showMessage(tr("Prêt"), 2000);
}
```

Ces slots ne sont pas très complexes. Ils mettent à jour la fenêtre (par exemple la barre de progression en bas) lorsqu'il y a lieu.

Certains sont très utiles, comme `changementUrl()`. En effet, lorsque l'utilisateur clique sur un lien sur la page, l'URL change et il faut par conséquent mettre à jour le champ d'adresse.

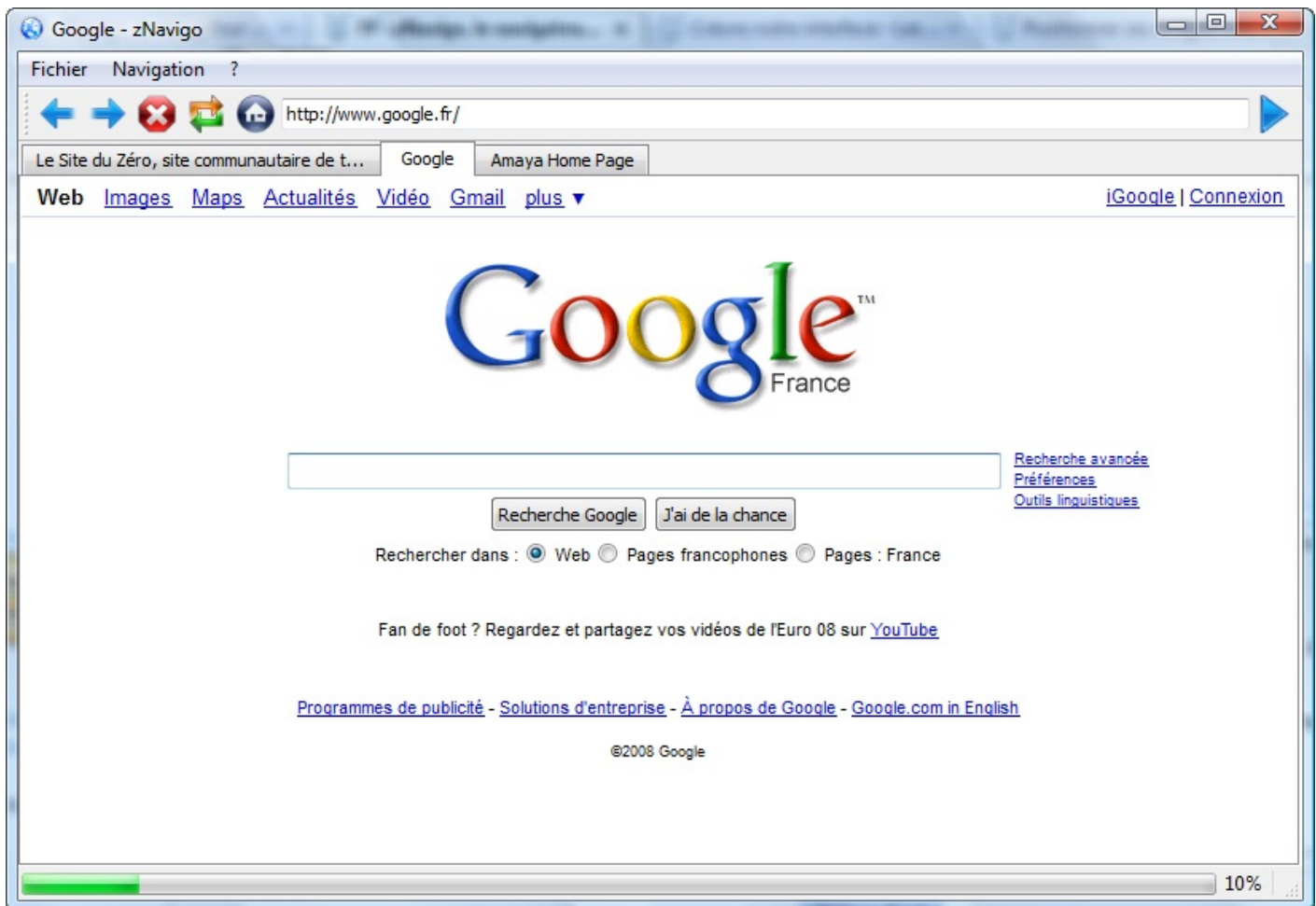
Vous noterez que je tronque le titre de la page à 40 caractères si celui-ci est trop long. Cela permet d'éviter d'avoir des onglets qui font 4km de large. 😊

Conclusion et améliorations possibles

Ouf! 😊

Pas fâché d'en avoir terminé, mais le plus beau dans tout ça, c'est qu'on a un navigateur parfaitement fonctionnel !

Je remets ici le screenshot par plaisir. 😊



A l'heure actuelle, un bug dans QtWebKit dans la gestion des cookies ne permet pas de se connecter au Site du Zéro. C'est un peu dommage, mais comme le module est assez récent il devrait s'améliorer et être corrigé à l'avenir. Ne soyez donc pas surpris si vous ne pouvez pas vous connecter au site avec.

Télécharger le code source et l'exécutable

Je vous propose de télécharger le code source ainsi que l'exécutable Windows du projet :

[Télécharger le code source et l'exécutable Windows \(39 Ko\)](#)

Pensez à ajouter les DLL nécessaires dans le même dossier que l'exécutable si vous voulez que celui-ci fonctionne. Cette fois, comme je vous l'avais dit, il faut 2 nouvelles DLL : QtWebKit4.dll et QtNetwork4.dll.

Améliorations possibles

Améliorer le navigateur, c'est possible ?

Certainement ! Il fonctionne, mais il est encore loin d'être parfait, et j'ai des tonnes d'idées pour l'améliorer. Bon ces idées sont repompées des navigateurs qui existent déjà, mais rien ne vous empêche d'en inventer de nouvelles super-révolutionnaires bien sûr. 🤖

- **Afficher l'historique dans un menu** : il existe une classe `QWebHistory` qui permet de récupérer l'historique de toutes les pages visitées via un `QWebView`. Pour obtenir un objet de type `QWebHistory` du `QWebView` de l'onglet en cours, utilisez `pageActuelle() -> page() -> history()`. Renseignez-vous ensuite sur la doc de `QWebHistory` pour essayer de trouver comment récupérer la liste des pages

visitées. Vous pourriez, par exemple, rajouter un menu "Historique" qui afficherait l'historique des pages vues sur l'onglet en cours.

- **Gestion des adresses HTTPS** : certains sites sont sécurisés (comme Paypal et Adsense par exemple). Ils utilisent des adresses `https://` au lieu de `http://`. A cause de la vérification que l'on a faite, notre navigateur n'accepte que les adresses `http://`. Modifiez-le pour qu'il puisse gérer aussi bien les `http://` que les `https://`.
- **Zone de recherche Google** : vous pourriez rajouter une zone de recherche google en haut à droite, qui appelle automatiquement Google avec les mots-clés sélectionnés.
Je vous laisse vous renseigner sur le fonctionnement de Google. Vous pouvez imiter un autre navigateur comme Firefox par exemple, pour voir l'URL qu'il charge lorsqu'on fait une recherche Google.
- **Recherche dans la page** : rajoutez la possibilité de faire une recherche dans le texte de la page web affichée. Indice : QWebView dispose d'une méthode `findText()` !
- **Disparition des onglets s'il ne reste plus qu'une seule page** : voilà une amélioration délicate. Un QTabWidget sans onglets ne peut exister. Mais s'il ne reste qu'une seule page affichée, ça ne sert à rien d'utiliser un système à onglets. Essayer de gérer le cas où il ne reste plus qu'une seule page affichée, afin que le QTabWidget disparaisse (au moins jusqu'à ce qu'on demande à ouvrir un nouvel onglet).
- **Fenêtre d'options** : vous pourriez créer une nouvelle fenêtre d'options qui permet de définir la taille de police par défaut, l'URL de la page d'accueil, etc.
Pour modifier la taille de la police par défaut, regardez du côté de [QWebSettings](#).

Pour enregistrer les options, vous pouvez passer par la classe QFile pour écrire dans un fichier. Mais j'ai mieux : utilisez la classe [QSettings](#) qui est spécialement faite pour enregistrer des options. En général, les options sont enregistrées dans un fichier (.ini, .conf...), mais on peut aussi enregistrer les options dans la base de registres sous Windows. Prenez bien le temps de lire la description de cette classe, c'est un peu long mais c'est vraiment très intéressant.

- **Gestion des marque-pages (favoris)** : voilà une fonctionnalité très répandue sur la plupart des navigateurs. L'utilisateur aime bien pouvoir enregistrer les adresses de ses sites web préférés.
Là encore, pour l'enregistrement, je vous recommande chaudement de passer par un [QSettings](#).

Vous pourrez ensuite afficher la liste des sites favoris dans un menu, ou encore dans une nouvelle barre d'outils comme le fait Firefox.

- **Impression d'une page web** : pourquoi ne pas faire s'amuser avec l'imprimante ?
Les QWebFrame contiennent une méthode [print\(\)](#) qui prend en paramètre une imprimante (QPrinter).

Le QWebView est constitué d'une QWebPage qui est elle-même constituée d'un ou plusieurs QWebFrame (généralement un seul). Je vous laisse découvrir comment obtenir un pointeur vers le QWebFrame.
Je vous laisse aussi découvrir comment manipuler les [QPrinter](#), qui permettent de faire appel à l'imprimante. Et je vous invite aussi à jeter un oeil à la classe [QPrintDialog](#) qui permet d'afficher une boîte de dialogue générique d'impression.

Ah, que de choses à découvrir !

- **Sauvegarde de l'état de la fenêtre à la clôture** : c'est peut-être une des fonctionnalités les plus appréciées des navigateurs actuels. Lorsque l'utilisateur veut quitter le programme, enregistrez (toujours avec QSettings) la liste des onglets ouverts avec leurs URL. Vous pourrez ainsi les réouvrir automatiquement lors du prochain chargement du programme.

Voilà, avec tout ce que je vous ai donné à faire, je crois que j'ai le temps d'aller à la nage à Hawaï siroter une Piña Colada dans un bar en bord de mer.

Et peut-être même que j'ai le temps de revenir d'ailleurs. 😊

L'architecture MVC avec les widgets complexes

Nous attaquons maintenant un des chapitres les plus intéressants de ce cours sur Qt, mais aussi un des plus difficiles.

Dans ce chapitre, nous apprendrons à manipuler 3 widgets complexes :

- **QListView** : une liste d'éléments à un seul niveau.
- **QTreeView** : une liste d'éléments à plusieurs niveaux, organisée en arbre.
- **QTableView** : un tableau.

On ne peut pas utiliser ces widgets sans un minimum de théorie. Et c'est justement cette partie théorique qui me fait dire que ce chapitre sera l'un des plus intéressants : nous allons découvrir l'architecture MVC, une façon de programmer (on parle de *design pattern*) très puissante qui va nous donner énormément de flexibilité.

Présentation de l'architecture MVC

Avant de commencer à manipuler les 3 widgets complexes dont je vous ai parlé en introduction, il est indispensable que je vous présente l'architecture MVC.



Qu'est-ce que l'architecture MVC ? A quoi ça sert ? Quel rapport avec la création de GUI ?

MVC est l'abréviation de Model-View-Controller, ce qui signifie en français : "Modèle-Vue-Contrôleur".

...

... ça ne vous avance pas trop, j'ai l'impression. 🤔

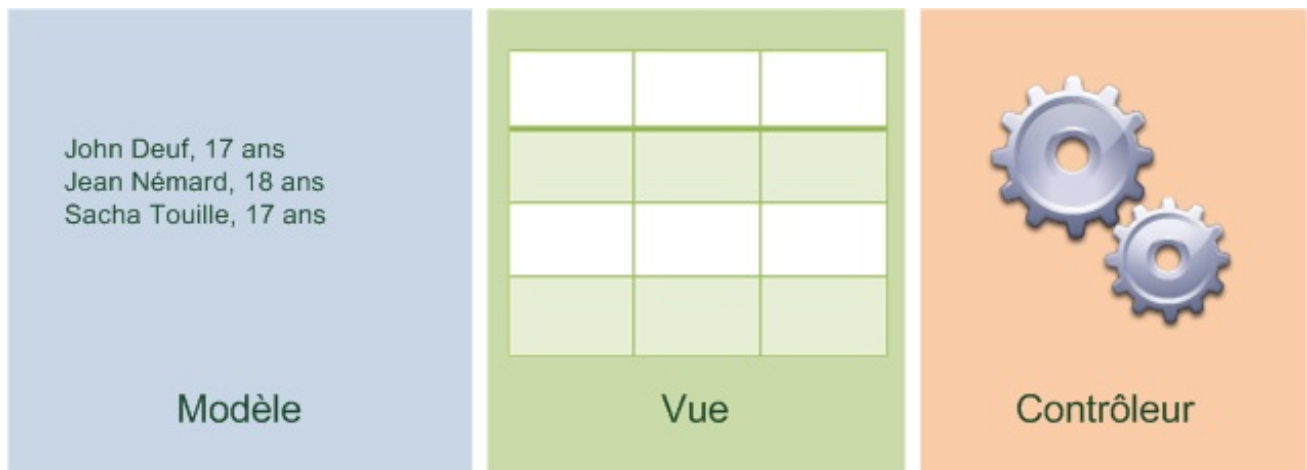
Il s'agit d'un *design pattern*, une technique de programmation. C'est une "façon de programmer et d'organiser son code" bien pensée. Vous n'êtes pas obligés de programmer de cette manière-là, mais si vous le faites votre code sera plus lisible, plus clair et plus souple.

L'architecture MVC vous propose de séparer les éléments de votre programme en 3 parties :

- **Le modèle** : c'est la partie qui contient les données. Le modèle peut par exemple contenir la liste des élèves d'une classe, avec leurs noms, prénoms, âges...
- **La vue** : c'est la partie qui s'occupe de l'affichage. Elle affiche ce que contient le modèle. Par exemple, la vue pourrait être un tableau. Ce tableau affichera la liste des élèves si c'est ce que contient le modèle.
- **Le contrôleur** : c'est la partie "réflexion" du programme. Lorsque l'utilisateur sélectionne 3 élèves dans le tableau et appuie sur la touche "Supprimer", le contrôleur est appelé et se charge de supprimer les 3 élèves du modèle.

C'est dur à imaginer au début. Mais rassurez-vous, vous allez comprendre au fur et à mesure de ce chapitre l'intérêt de séparer le code en 3 parties et le rôle de chacune de ces parties. Ce n'est pas grave si vous ne voyez pas de suite comment ces parties interagissent entre elles.

Commençons par un schéma, visuel et simple à retenir, qui présente le rôle de chacune de ces parties :



Comme on peut le voir sur ce schéma :

- **Le modèle** est la partie qui contient les données (comment, on verra ça après). Les données sont généralement récupérées en lisant un fichier ou une base de données.
- **La vue** est juste la partie qui affiche le modèle, ce sera donc un widget dans notre cas. Si un élément est ajouté au modèle (par exemple un nouvel élève apparaît) la vue se met à jour automatiquement pour afficher le nouveau modèle.
- **Le contrôleur** est la partie la plus algorithmique, c'est-à-dire le cerveau de votre programme. S'il y a des calculs à faire, c'est là qu'ils sont faits.

L'architecture simplifiée modèle/vue de Qt

En fait (vous allez me tuer je le sens 🤔), Qt n'utilise pas vraiment MVC mais une version simplifiée de ce système : l'architecture modèle/vue.



Et le contrôleur on en fait quoi ? Poubelle ? Notre programme ne réfléchit pas ?

Si si, je vous rassure. En fait, le contrôleur est intégré à la vue avec Qt.

Grâce à ça, les données sont toujours séparées de leur affichage, mais on diminue un peu la complexité du modèle MVC en évitant au programmeur d'avoir à gérer les 3 parties.

On s'éloigne donc un petit peu de la théorie pure sur MVC ici, pour s'intéresser à la façon dont Qt utilise ce principe en pratique. Vous venez donc d'apprendre que Qt adapte ce principe à sa manière, pour garder les bonnes idées principales sans pour autant vous obliger à "trop" découper votre code ce qui aurait pu être un peu trop complexe et répétitif à la longue.

Nous n'allons donc plus parler ici que de modèle et de vue. Comment sont gérés chacun de ces éléments avec Qt ?

Cette question... avec des classes, comme d'habitude ! 🤖

Les classes gérant le modèle

Il y a plusieurs types de modèles différents. En effet : on ne stocke pas de la même manière une liste d'élèves qu'une liste de villes !

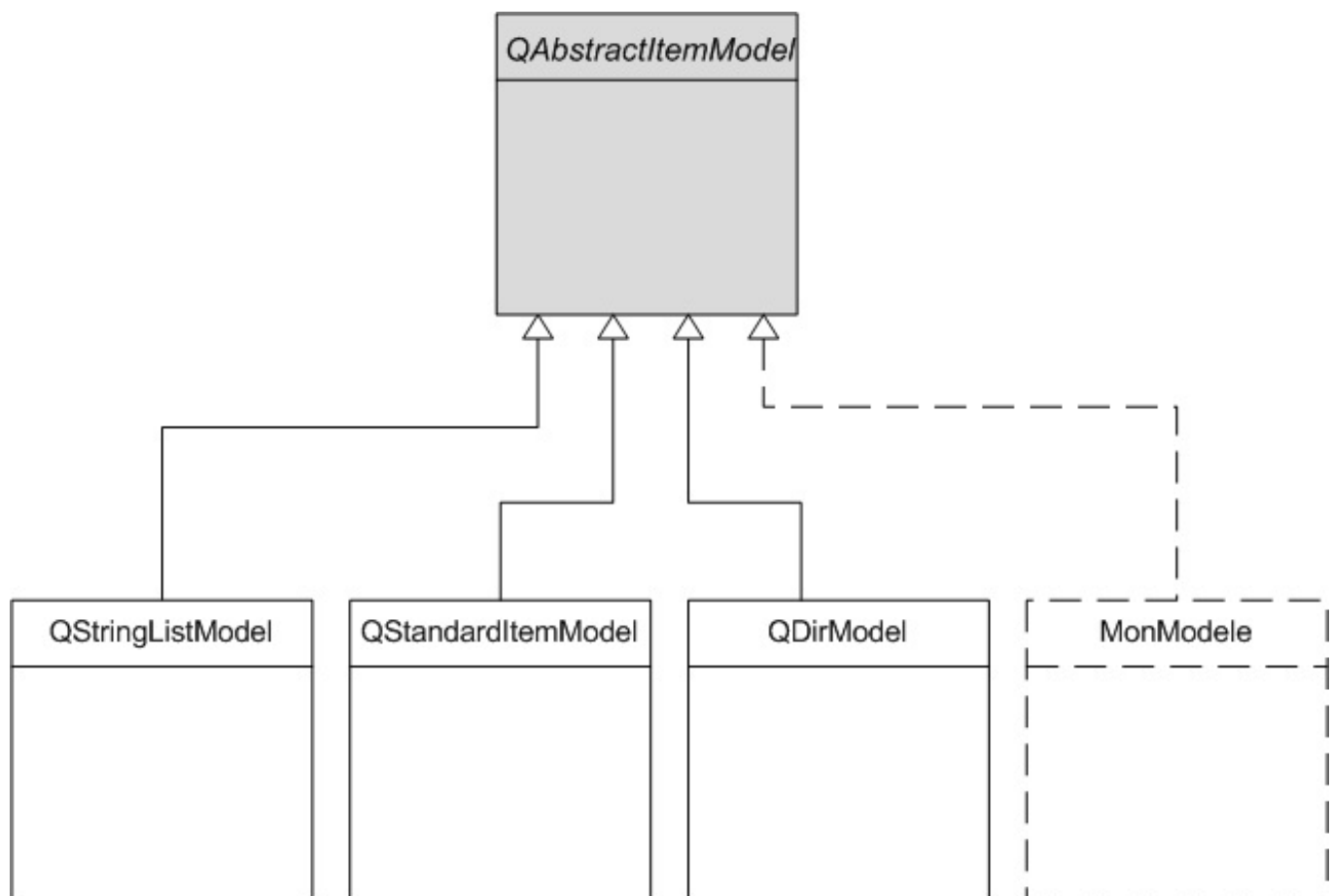
Vous avez 2 possibilités :

- Soit vous créez **votre propre classe de modèle**. Il faut créer une classe héritant de `QAbstractItemModel`. C'est la solution la plus flexible mais aussi la plus complexe, nous ne la verrons pas ici.
- Soit vous utilisez une des **classes génériques toutes prêtes** offertes par Qt :
 - `QStringListModel` : une liste de chaînes de caractères, de type `QString`. Très simple, très basique. Ca peut suffire pour les cas les plus simples.

- **QStandardItemModel** : une liste d'éléments organisés sous forme d'arbre (chaque élément peut contenir des sous-éléments). Ce type de modèle est plus complexe que le précédent, car il gère plusieurs niveaux d'éléments. Avec **QStringListModel**, c'est un des modèles les plus utilisés.
- **QDirModel** : la liste des fichiers et dossiers stockés sur votre ordinateur. Ce modèle va analyser en arrière-plan votre disque dur, et restitue la liste de vos fichiers sous la forme d'un modèle prêt à l'emploi.
- **QSqlQueryModel**, **QSqlTableModel** et **QSqlRelationalTableModel** : données issues d'une base de données. On peut s'en servir pour accéder à une base de données (ceux qui ont déjà utilisé MySQL, Oracle ou un autre système du genre seront probablement intéressés).
Je ne vais pas rentrer dans les détails de la connexion à une base de données dans ce chapitre, ce serait un peu hors-sujet.

Toutes ces classes proposent donc des modèles prêts à l'emploi, qui héritent de **QAbstractItemModel**.

Si aucune de ces classes ne vous convient, vous devrez créer votre propre classe en héritant de **QAbstractItemModel**.



Notez que je n'ai pas représenté toutes les classes de modèles ici.

Les classes gérant la vue

Pour afficher les données issues du modèle, il nous faut une vue. Avec Qt, la vue est un widget, puisqu'il s'agit d'un affichage dans une fenêtre.

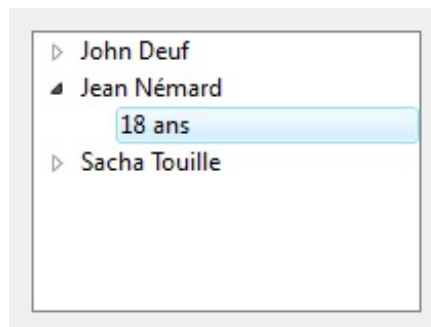
Tous les widgets de Qt ne sont pas bâtis autour de l'architecture modèle/vue, loin de là (et ça n'aurait pas d'intérêt pour les plus simples d'entre eux que nous avons vu jusqu'à présent).

On compte 3 widgets adaptés pour la vue avec Qt :

- **QListView** : une liste d'éléments.



- **QTreeView** : un arbre d'éléments, où chaque élément peut avoir des éléments enfants.



- **QTableView** : un tableau.

John	Deuf	17 ans
Jean	Némard	18 ans
Sacha	Touille	17 ans

Voilà donc les fameux "widgets" complexes que je vais vous présenter dans ce chapitre. Mais pour pouvoir les utiliser et les peupler de données, il faut d'abord créer un modèle !

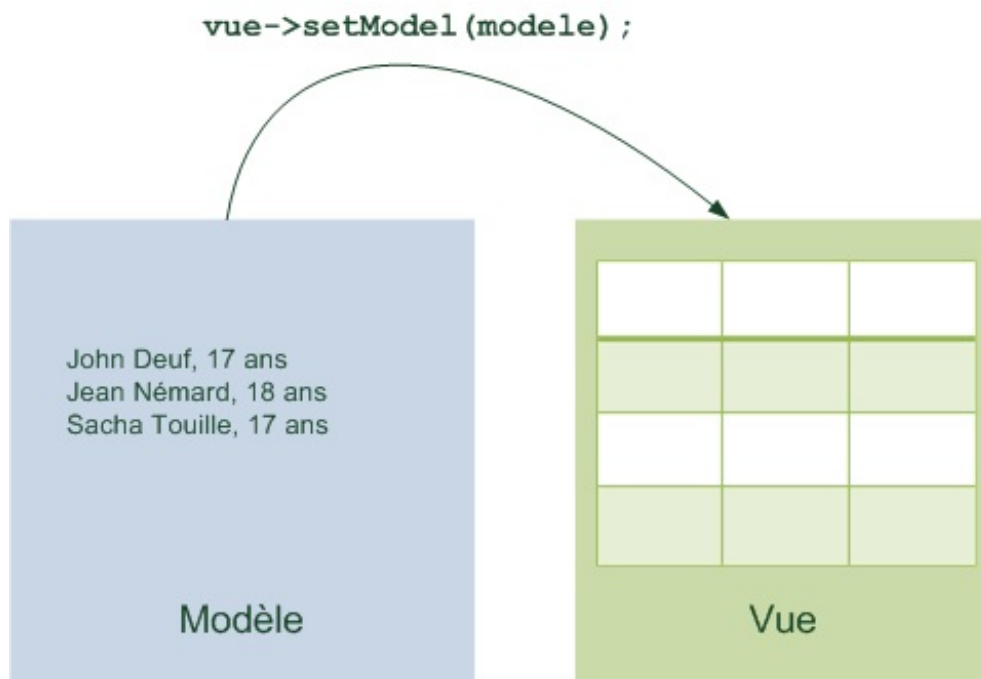
Appliquer un modèle à la vue

Lorsqu'on utilise l'architecture modèle/vue avec Qt, cela se passe toujours en 3 temps. Il faut :

1. Créer le modèle
2. Créer la vue
3. Associer la vue et le modèle

La dernière étape est essentielle. Cela revient en quelque sorte à "connecter" notre modèle à notre vue. Si on ne donne pas de modèle à la vue, elle ne saura pas quoi afficher, donc elle n'affichera rien. 🤪

La connexion se fait toujours avec la méthode `setModel()` de la vue :



Le contrôleur n'a pas été représenté sur ce schéma car, comme je vous l'ai dit, Qt utilise une architecture modèle/vue simplifiée et se charge de gérer la partie contrôleur pour vous.

Voilà donc comment on connecte un modèle à une vue en pratique. 😊

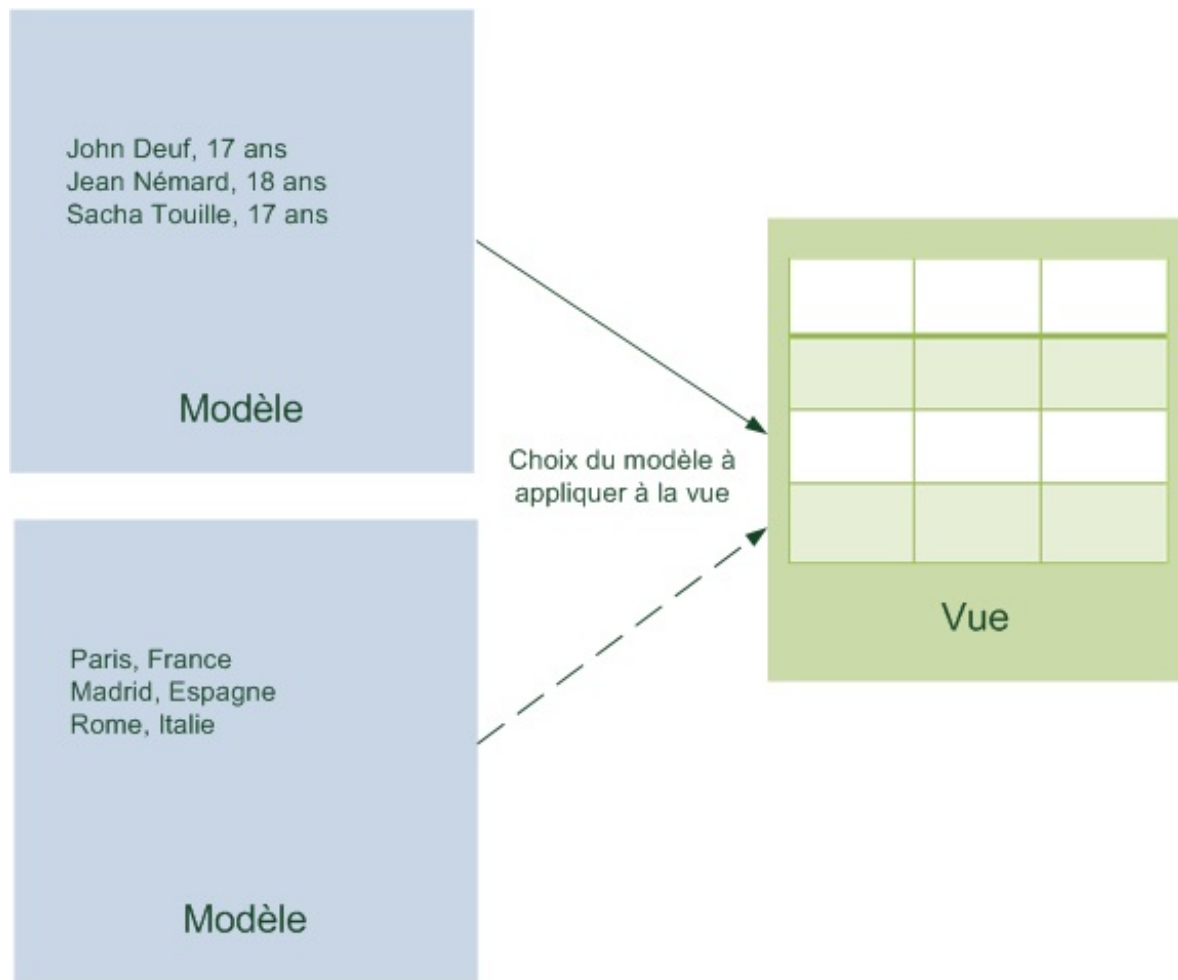
L'avantage de ce système, c'est qu'il est flexible. On peut avoir 2 modèles et appliquer soit l'un soit l'autre à la vue en fonction des besoins. Un gros intérêt est que dès que l'on modifie le modèle, la vue affiche instantanément les nouveaux éléments !

Plusieurs modèles ou plusieurs vues

On peut pousser le principe un peu plus loin. Essayons d'imaginer que l'on a plusieurs modèles ou plusieurs vues.

Plusieurs modèles et une vue

Imaginons que l'on ait 2 modèles : un qui contient une liste d'élèves, un autre qui contient une liste de capitales avec leur pays. Notre vue peut afficher soit l'un, soit l'autre :

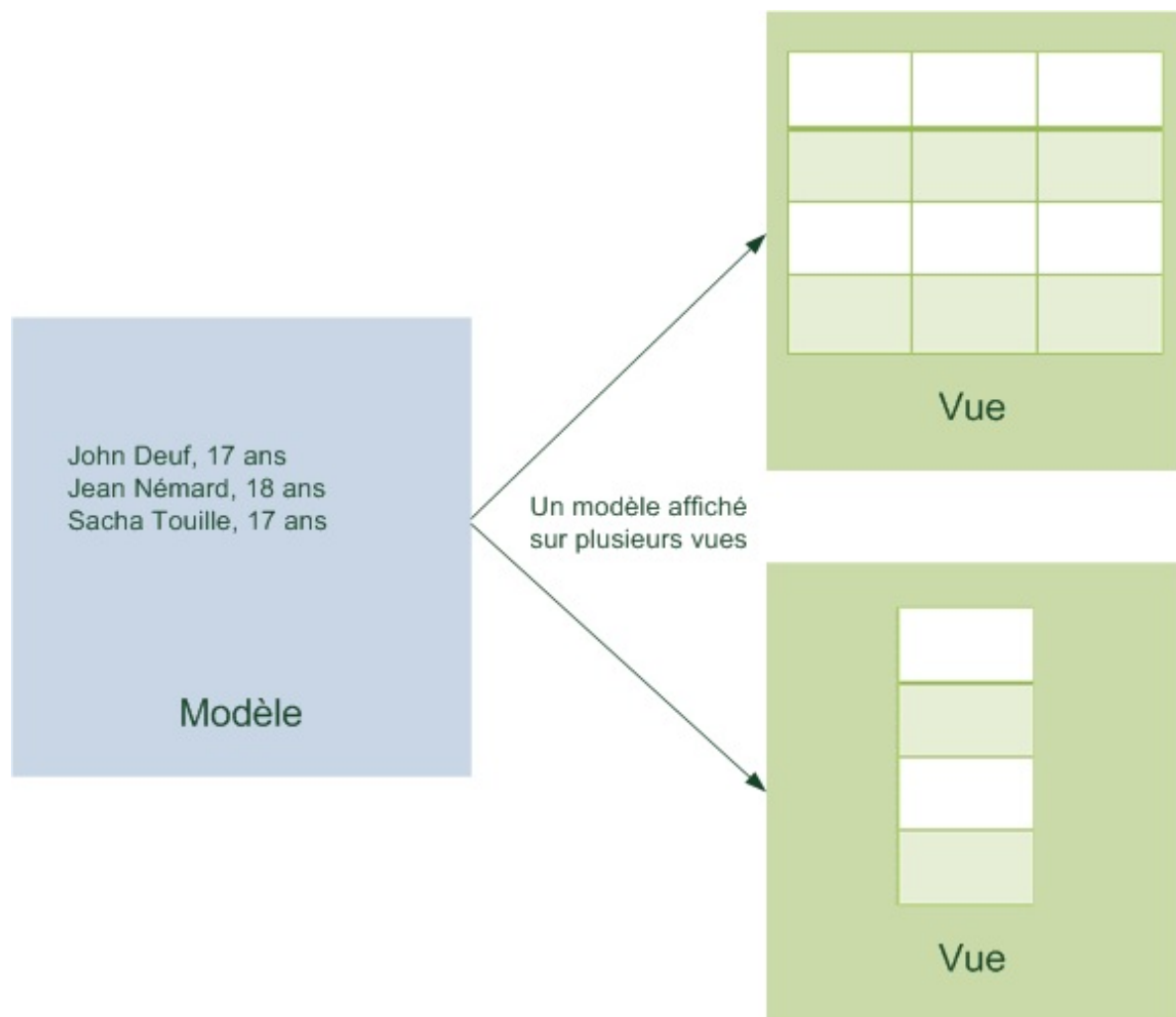


Il faut bien sûr faire un choix : une vue ne peut afficher qu'un seul modèle à la fois !

L'avantage, c'est qu'au besoin on peut changer le modèle affiché par la vue en cours d'exécution, en appelant juste la méthode `setModel()` !

Un modèle pour plusieurs vues

Imaginons le cas inverse. On a un modèle, mais plusieurs vues. Cette fois, rien ne nous empêche d'appliquer ce modèle à 2 vues en même temps !



On peut ainsi visualiser le même modèle de 2 façons différentes (ici sous forme de tableau ou de liste dans mon schéma). Comme le même modèle est associé à 2 vues différentes, si le modèle change alors les 2 vues changent en même temps ! Par exemple, si je modifie l'âge d'un des élèves dans une cellule du tableau, l'autre vue (la liste) est automatiquement mise à jour sans avoir besoin d'écrire la moindre ligne de code !

Utilisation d'un modèle simple

Pour découvrir en douceur l'architecture modèle/vue de Qt, je vais vous proposer d'utiliser un modèle tout fait : [QDirModel](#).

Sa particularité, c'est qu'il est très simple à utiliser. Il analyse votre disque dur et génère le modèle correspondant. Pour créer ce modèle, c'est tout bête :

Code : C++

```
QDirModel *modele = new QDirModel;
```

On possède désormais un modèle qui représente notre disque. On va l'appliquer à une vue.



Mais quelle vue utiliser ? Une liste, un arbre, un tableau ? Les modèles sont-ils compatibles avec toutes les vues ?

Oui, toutes les vues peuvent afficher n'importe quel modèle. C'est toujours compatible.

Par contre, même si ça marche avec toutes les vues, vous allez vous rendre compte que certaines sont plus adaptées que d'autres en fonction du modèle que vous utilisez.

Par exemple, pour un `QDirModel`, la vue la plus adaptée est sans aucun doute l'arbre (`QTreeView`). Nous essaierons toutefois toutes les vues avec ce modèle pour comparer le fonctionnement.

Le modèle appliqué à un QTreeView

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

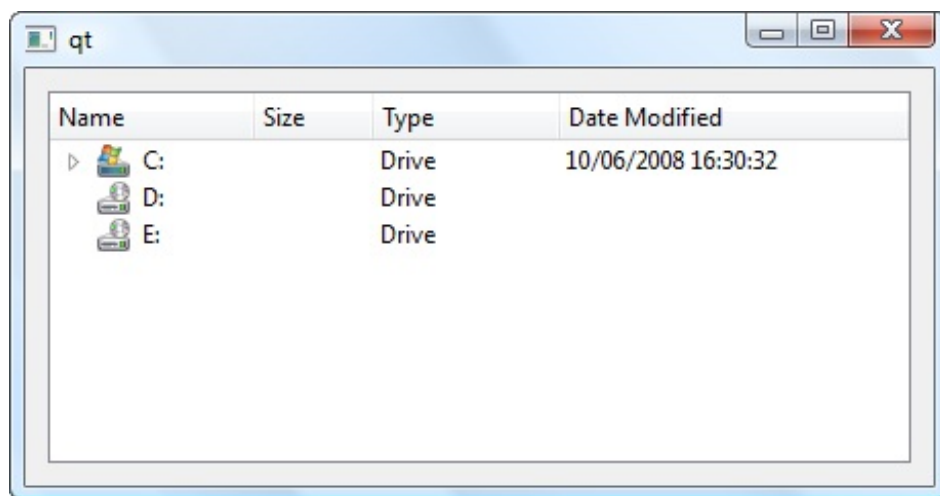
    QDirModel *modele = new QDirModel;
    QTreeView *vue = new QTreeView;
    vue->setModel(modele);

    layout->addWidget(vue);

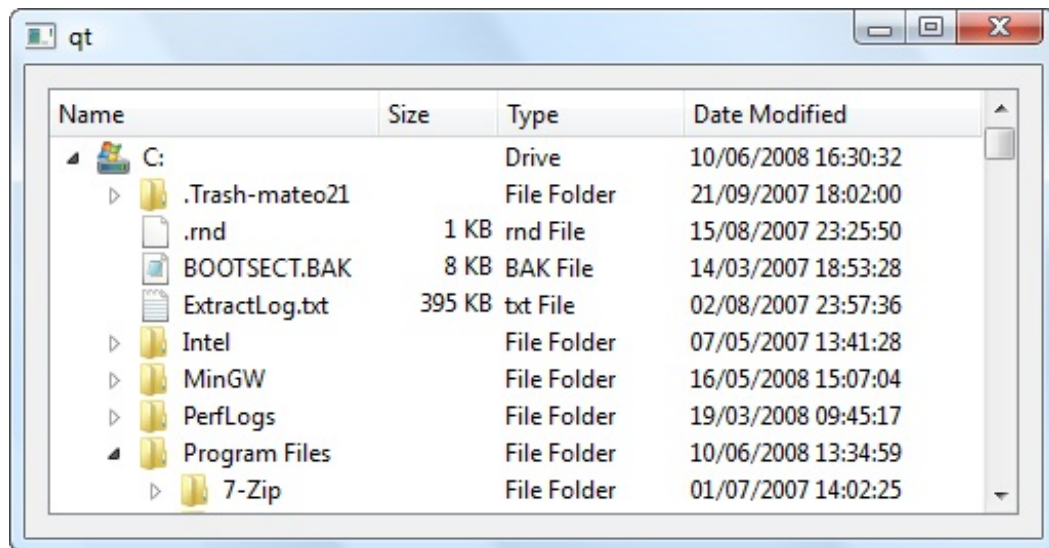
    setLayout(layout);
}
```

On crée le modèle, puis la vue, et on dit à la vue "Utilise ce modèle pour savoir quoi afficher" (ligne 9).

Le résultat est le suivant :



Une vue en forme d'arbre affiche le modèle de notre disque. Chaque élément peut avoir des sous-éléments dans un QTreeView, essayez de naviguer dedans :



Voilà un bel exemple d'arbre en action ! 😊

Le modèle appliqué à un QListView

Maintenant, essayons de faire la même chose, mais avec une liste (QListView). On garde le même modèle, mais on l'applique à une vue différente :

Code : C++

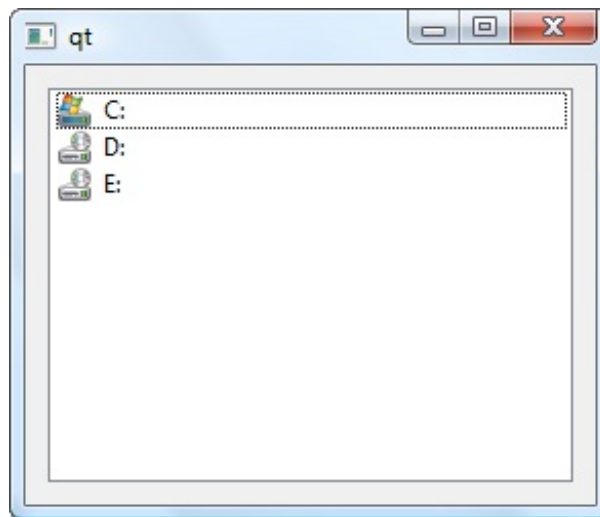
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QDirModel *modele = new QDirModel;
    QListView *vue = new QListView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```



Ce type de vue ne peut afficher qu'un seul niveau d'éléments à la fois. Cela explique pourquoi je vois uniquement la liste de mes disques... et pourquoi je ne peux pas afficher leur contenu !

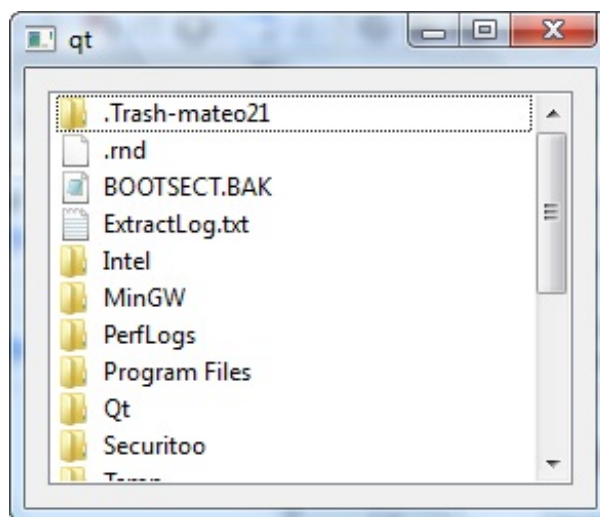
La vue affiche bêtement ce qu'elle est capable d'afficher, c'est-à-dire le premier niveau d'éléments.

Voilà la preuve qu'un même modèle marche sur plusieurs vues différentes, mais que certaines vues sont plus adaptées que d'autres !

Vous pouvez modifier la racine utilisée par la vue en vous inspirant du code suivant, que je ne détaillerai pas pour le moment :

Code : C++

```
vue->setRootIndex (modele->index ("C:")) ;
```



Le modèle appliqué à un QTableView

Un tableau ne peut pas afficher plusieurs niveaux d'éléments (seul l'arbre QTreeView peut le faire). Par contre, il peut afficher plusieurs colonnes :

Code : C++

```
#include "FenPrincipale.h"
```

```

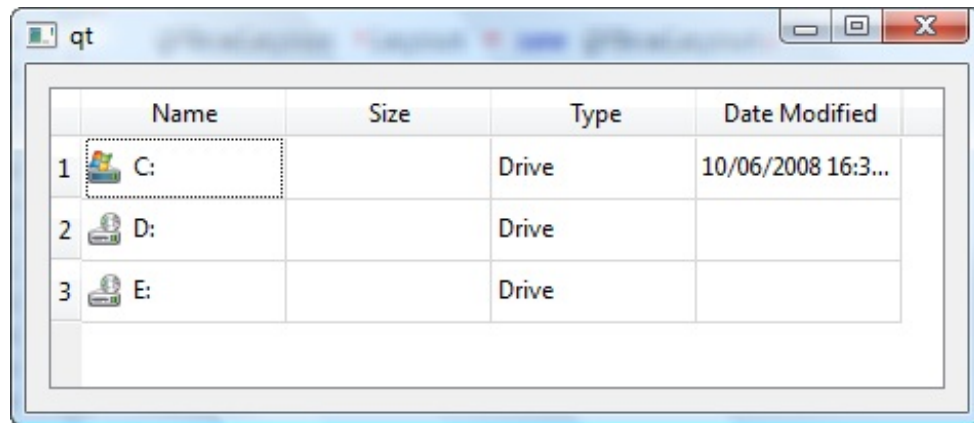
FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QDirModel *modele = new QDirModel;
    QTableView *vue = new QTableView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}

```



Comme précédemment, on peut appeler `setRootIndex()` pour modifier la racine des éléments affichés par la vue.

Utilisation de modèles personnalisables

Le modèle `QDirModel` que nous venons de voir était très simple à utiliser. Rien à paramétrer, rien à configurer, il analyse automatiquement votre disque dur pour construire son contenu.

C'était une bonne introduction pour découvrir les vues avec un modèle simple. Cependant, dans la plupart des cas, vous voudrez utiliser vos propres données, votre propre modèle. C'est ce que nous allons voir ici, à travers 2 nouveaux modèles :

- `QStringListModel` : une liste simple d'éléments de type texte, à un seul niveau.
- `QStandardItemModel` : une liste plus complexe à plusieurs niveaux et plusieurs colonnes, qui peut convenir dans la plupart des cas.

Pour les cas simples, nous utiliserons donc `QStringListModel`, mais nous découvrirons aussi `QStandardItemModel` qui nous donne plus de flexibilité.

QStringListModel : une liste de chaînes de caractères QString

Ce modèle, très simple, vous permet de gérer une liste de chaînes de caractères. Par exemple, si l'utilisateur doit choisir son pays parmi une liste :

France
Espagne
Italie
Portugal
Suisse

Pour construire ce modèle, il faut procéder en 2 temps :

- Construire un objet de type `QStringList`, qui contiendra la liste des chaînes.

- Créer un objet de type `QStringListModel` et envoyer à son constructeur le `QStringList` que vous venez de créer pour l'initialiser.

`QStringList` surcharge l'opérateur "<<" pour vous permettre d'ajouter des éléments à l'intérieur simplement. Un exemple de code sera sûrement plus parlant :

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

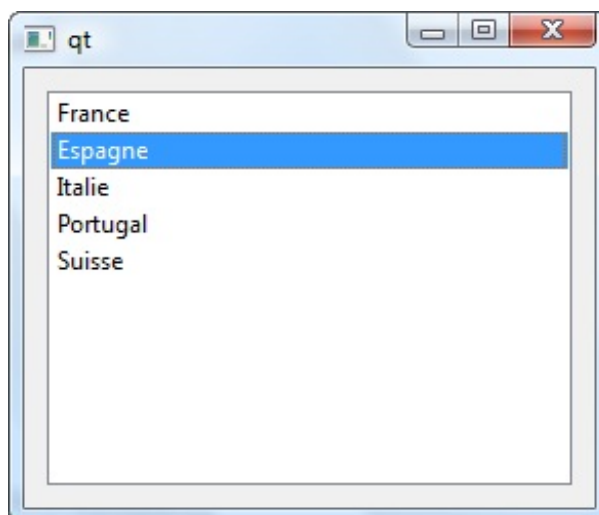
    QStringList listePays;
    listePays << "France" << "Espagne" << "Italie" << "Portugal" <<
    "Suisse";
    QStringListModel *modele = new QStringListModel(listePays);

    QListView *vue = new QListView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

Notre vue affiche maintenant la liste des pays qui se trouvent dans le modèle :



La surcharge de l'opérateur "<<" est très pratique comme vous pouvez le voir. Sachez toutefois qu'il est aussi possible d'utiliser la méthode `append()` :

Code : C++

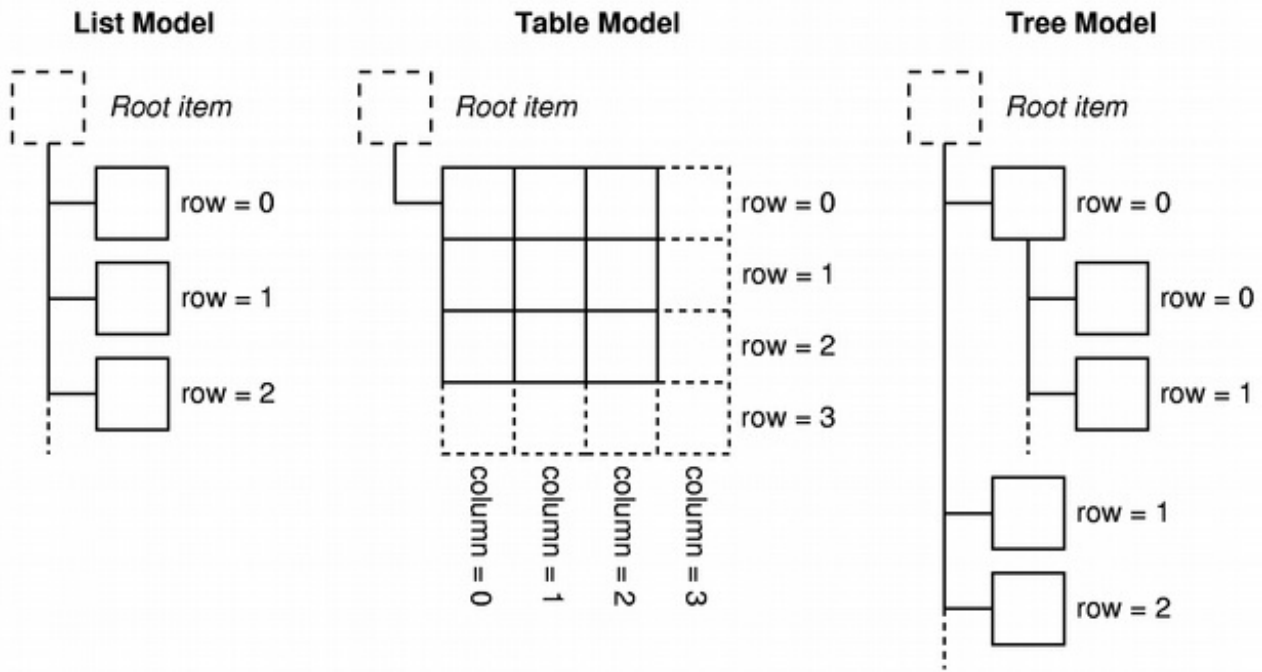
```
listePays.append("Belgique");
```

Si, au cours de l'exécution du programme, un pays est ajouté, supprimé ou modifié, la vue (la liste) affichera automatiquement les modifications. Nous testerons cela en pratique un peu plus loin dans le chapitre.

QStandardItemModel : une liste à plusieurs niveaux et plusieurs colonnes

Ce type de modèle est beaucoup plus complet (et donc complexe 😊) que le précédent. Il permet de créer tous les types de modèles possibles et imaginables.

Pour bien visualiser les différents types de modèles que l'on peut concevoir avec un QStandardItemModel, voici un super schéma offert par la doc de Qt :



- **List Model** : c'est un modèle avec une seule colonne et pas de sous-éléments. C'est le modèle utilisé par QStringList, mais QStandardItemModel peut aussi le faire (qui peut le plus peut le moins !).
Ce type de modèle est en général adapté à un QListView.
- **Table Model** : les éléments sont organisés avec plusieurs lignes et colonnes.
Ce type de modèle est en général adapté à un QTableView.
- **Tree Model** : les éléments ont des sous-éléments, ils sont organisés en plusieurs niveaux. Ce n'est pas représenté sur le schéma ci-dessus, mais rien n'interdit de mettre **plusieurs colonnes dans un modèle en arbre** aussi !
Ce type de modèle est en général adapté à un QTreeView.

Gérer plusieurs lignes et colonnes

Pour construire un QStandardItemModel, on doit indiquer en paramètres le nombre de lignes et de colonnes qu'il doit gérer. Des lignes et des colonnes supplémentaires peuvent toujours être ajoutées par la suite au besoin.

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStandardItemModel *modele = new QStandardItemModel(5, 3);

    QTableView *vue = new QTableView;
    vue->setModel(modele);
```



```

        layout->addWidget (vue);

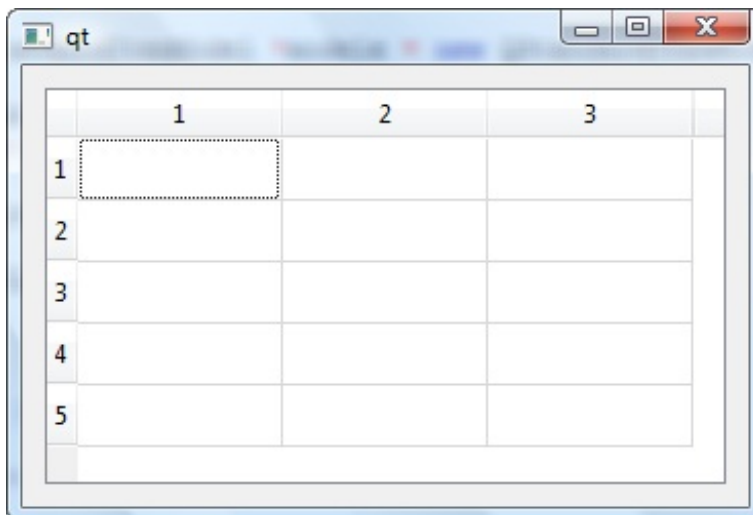
        setLayout (layout);
    }

```



Si on ne demande qu'une seule colonne, cela reviendra à créer un modèle de type "List Model".

Ici, un modèle à 5 lignes et 3 colonnes sera créé. Les éléments sont vides au départ, mais on a déjà un tableau :



On peut aussi appeler le constructeur par défaut (sans paramètres) si on ne connaît pas du tout la taille du tableau à l'avance.

Il faudra appeler `appendRow()` et `appendColumn()` pour ajouter respectivement une nouvelle ligne ou une nouvelle colonne.

Chaque élément est représenté par un objet de type `QStandardItem`.

Pour définir un élément, on utilise la méthode `setItem()` du modèle. Donnez-lui respectivement le numéro de ligne, de colonne, et un `QStandardItem` à afficher. Attention : la numérotation commence à 0.

Code : C++

```

#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStandardItemModel *modele = new QStandardItemModel(5, 3);
    modele->setItem(3, 1, new QStandardItem("Zéro !"));

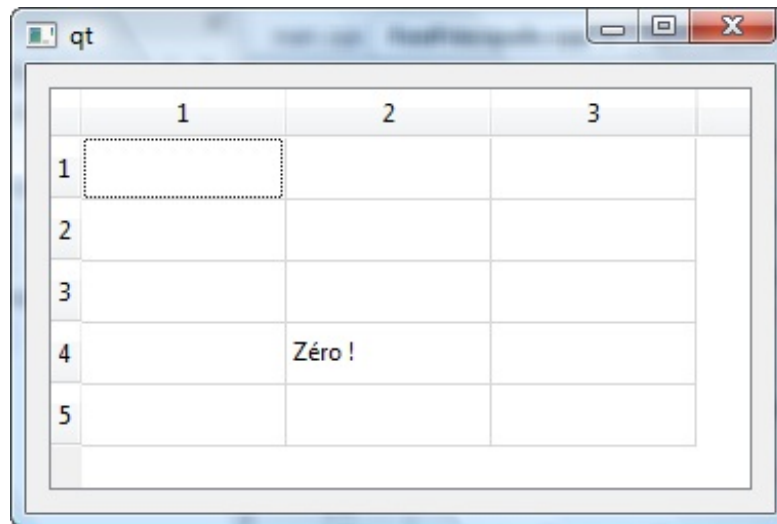
    QTableView *vue = new QTableView;
    vue->setModel (modele);

    layout->addWidget (vue);

    setLayout (layout);
}

```

Ici, je crée un nouvel élément contenant le texte "Zéro !" à la 4ème ligne, 2nde colonne :



Voilà comment on peut peupler le modèle d'éléments. 😊

Ajouter des éléments enfants

Essayons maintenant d'ajouter des éléments enfants.

Pour pouvoir voir les éléments enfants, on va devoir changer de vue et passer par un QTreeView.

Il faut procéder dans l'ordre :

1. Créer un élément (par exemple "item"), de type QStandardItem. *Ligne 9*
2. Ajouter cet élément **au modèle** avec appendRow(). *Ligne 10*
3. Ajouter un sous-élément à **"item"** avec appendRow(). *Ligne 11*

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStandardItemModel *modele = new QStandardItemModel;

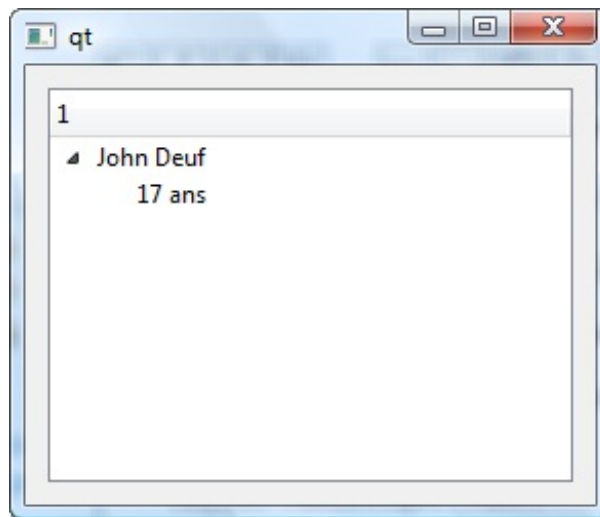
    QStandardItem *item = new QStandardItem("John Deuf");
    modele->appendRow(item);
    item->appendRow(new QStandardItem("17 ans"));

    QTreeView *vue = new QTreeView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

Résultat, on a créé un élément "John Deuf" qui contient un élément enfant "17 ans" :



Entraînez-vous à créer plusieurs éléments et des sous-éléments enfants, ce n'est pas compliqué si on est bien organisé mais il faut pratiquer. 😊



Pour supprimer l'en-tête (marqué "1" et inutile), vous pouvez appeler : `vue->header()->hide();`

Gestion des sélections

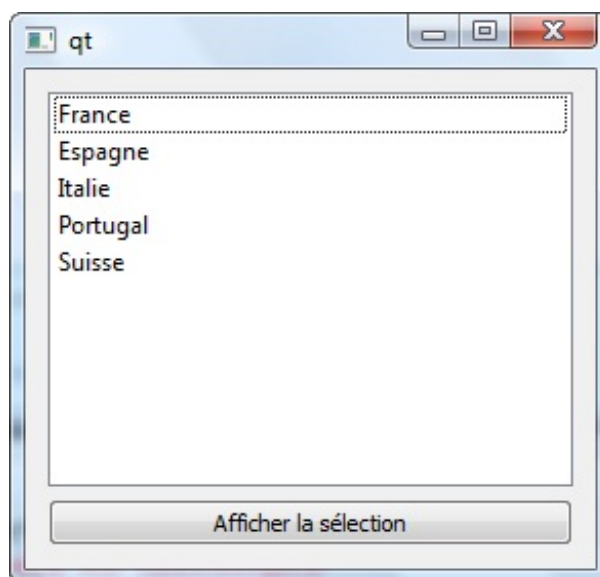
Nous avons découvert comment associer un modèle à une vue, et comment manipuler plusieurs modèles : `QDirModel`, `QStringListModel` et `QStandardItemModel`.

Il nous reste à voir comment on peut récupérer le ou les éléments sélectionnés dans la vue, pour savoir quel est le choix de l'utilisateur.

Nous entrons dans une partie vraiment complexe où de nombreuses classes se mêlent les unes aux autres. L'architecture modèle/vue de Qt est extrêmement flexible (on peut faire ce qu'on veut avec), mais en contrepartie il est beaucoup plus délicat de s'en servir car il y a plusieurs étapes à suivre dans un ordre précis.

Par conséquent, et afin d'éviter de faire un chapitre beaucoup trop long et surtout trop complexe, j'ai volontairement décidé de limiter mes exemples ici aux **sélections d'un `QListView`**. Je vous laisserai le soin d'adapter ces exemples aux autres vues, en sachant que c'est relativement similaire à chaque fois (les principes sont les mêmes).

Nous allons rajouter un bouton "Afficher la sélection" à notre fenêtre. Elle va ressembler à ceci :



Lorsqu'on cliquera sur le bouton, il ouvrira une boîte de dialogue (`QMessageBox`) qui affichera le nom de l'élément sélectionné.

Nous allons apprendre à gérer 2 cas :

- Lorsqu'on ne peut sélectionner qu'**un seul élément** à la fois.
- Lorsqu'on peut sélectionner **plusieurs éléments** à la fois.

Une sélection unique

Nous allons devoir créer une connexion entre un signal et un slot pour que le clic sur le bouton fonctionne.

Modifions donc pour commencer le .h de la fenêtre :

Code : C++

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QWidget
{
    Q_OBJECT

public:
    FenPrincipale();

private:
    QListView *vue;
    QStringListModel *modele;
    QPushButton *bouton;

private slots:
    void clicSelection();
};

#endif
```

J'ai rajouté la macro `Q_OBJECT`, mis quelques éléments de la fenêtre en attributs (pour pouvoir y accéder dans le slot), et ajouté le slot `clicSelection()` qui sera appelé après un clic sur le bouton.

Maintenant retour au .cpp, où je fais la connexion et où j'écris le contenu du slot :

Code : C++

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStringList listePays;
    listePays << "France" << "Espagne" << "Italie" << "Portugal" <<
    "Suisse";
    modele = new QStringListModel(listePays);

    vue = new QListView ;
    vue->setModel(modele);

    bouton = new QPushButton("Afficher la sélection");

    layout->addWidget(vue);
    layout->addWidget(bouton);
```

```

        setLayout(layout);

        connect(bouton, SIGNAL(clicked()), this, SLOT(clicSelection()));
    }

    void FenPrincipale::clicSelection()
    {
        QItemSelectionModel *selection = vue->selectionModel();
        QModelIndex indexElementSelectionne = selection->currentIndex();
        QVariant elementSelectionne = modele->data(indexElementSelectionne,
        Qt::DisplayRole);
        QMessageBox::information(this, "Elément sélectionné",
        elementSelectionne.toString());
    }

```

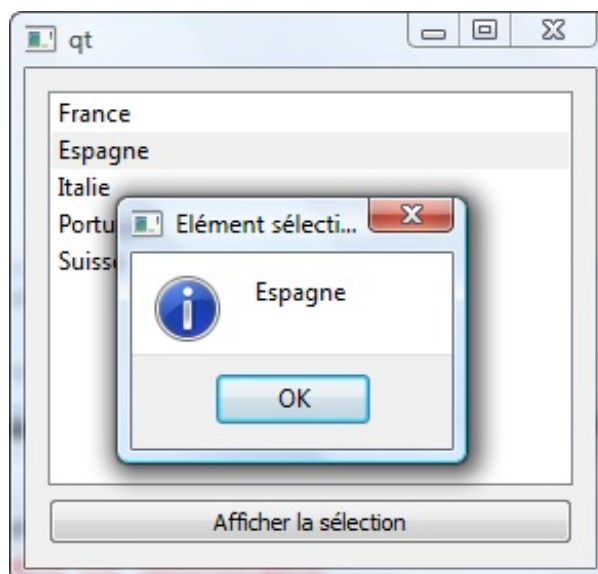
Analysons le contenu du slot, qui contient les nouveautés (lignes 26 à 29). Voici ce que nous faisons ligne par ligne :

1. On récupère un objet `QItemSelectionModel` qui contient des informations sur ce qui est sélectionné sur la vue. C'est la vue qui nous donne un pointeur vers cet objet grâce à `vue->selectionModel()`.
2. On appelle la méthode `currentIndex()` de l'objet qui contient des informations sur la sélection. Cela renvoie un index, c'est-à-dire en gros le numéro de l'élément sélectionné sur la vue.
3. Maintenant qu'on connaît le numéro de l'élément sélectionné, on veut retrouver son texte. On appelle la méthode `data()` du modèle, et on lui donne l'index qu'on a récupéré (c'est-à-dire le numéro de l'élément sélectionné). On récupère le résultat dans un `QVariant`, qui est une classe qui peut aussi bien stocker des int que des chaînes de caractères.
4. On n'a plus qu'à afficher l'élément sélectionné récupéré. Pour extraire la chaîne du `QVariant`, on appelle `toString()`.

Ouf ! Ce n'est pas simple je le reconnais, il y a plusieurs étapes.

D'abord on récupère l'objet qui contient des informations sur les éléments sélectionnés sur la vue. Ensuite on demande à cet objet quel est l'indice (numéro) de l'élément actuellement sélectionné. On peut alors récupérer le texte contenu dans le modèle à cet indice. On affiche ce texte avec la méthode `toString()` de l'objet de type `QVariant`.

Désormais, un clic sur le bouton vous indique quel élément est sélectionné :



Une sélection multiple

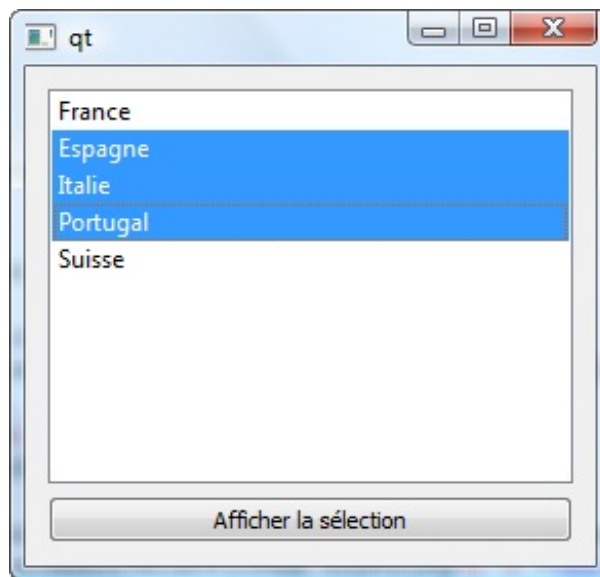
Par défaut, on ne peut sélectionner qu'un seul élément à la fois sur une liste. Pour changer ce comportement et autoriser la sélection multiple, rajoutez ceci dans le constructeur :

Code : C++

```
vue->setSelectionMode(QAbstractItemView::ExtendedSelection);
```

D'autres modes de sélection sont disponibles, mais je vous laisse aller voir la doc de QAbstractItemView pour en savoir plus. 😊
Avec ce mode, on peut sélectionner n'importe quels éléments. On peut utiliser la touche Shift du clavier pour faire une sélection continue, ou Ctrl pour une sélection discontinue (avec des trous).

Voici un exemple de sélection continue, désormais possible :



Pour récupérer la liste des éléments sélectionnés, c'est un peu plus compliqué ici parce qu'il y en a plusieurs. On ne peut plus utiliser la méthode currentIndex(), il va falloir utiliser selectedIndexes().

Je vous donne le nouveau code du slot, et on l'analyse ensuite ensemble. 😊

Code : C++

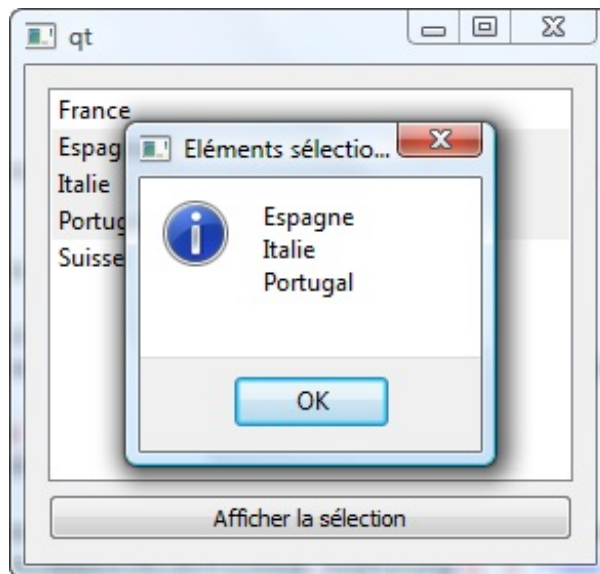
```
void FenPrincipale::clicSelection()
{
    QItemSelectionModel *selection = vue->selectionModel();
    QModelIndexList listeSelections = selection->selectedIndexes();
    QString elementsSelectionnes;

    for (int i = 0 ; i < listeSelections.size() ; i++)
    {
        QVariant elementSelectionne = modele-
>data(listeSelections[i], Qt::DisplayRole);
        elementsSelectionnes += elementSelectionne.toString() + "<br
/>";
    }

    QMessageBox::information(this, "Éléments sélectionnés",
elementsSelectionnes);
}
```

C'est un peu plus gros bien sûr. 😊

1. La première ligne ne change pas : on récupère l'objet qui contient des informations sur les éléments sélectionnés.
2. Ensuite, au lieu d'appeler `currentIndex()`, on demande `selectedIndexes()` parce qu'il peut y en avoir plusieurs.
3. On crée un `QString` vide dans lequel on stockera la liste des pays pour l'afficher ensuite dans la boîte de dialogue.
4. Vient ensuite une boucle. En effet, l'objet `listeSelections` récupéré est un tableau (en fait c'est un objet de type `QList`, mais on peut faire comme si c'était un tableau). On parcourt donc ce tableau ligne par ligne, et on récupère à chaque fois le texte correspondant.
5. On stocke ce texte à la suite du `QString`, qui se remplit au fur et à mesure.
6. Une fois la boucle terminée, on affiche le `QString` qui contient la liste des pays sélectionnés. Et voilà le travail !



Ici, je me contente d'afficher la liste des éléments sélectionnés dans une boîte de dialogue, mais en pratique vous ferez sûrement quelque chose de beaucoup plus intelligent avec ça. 🤔

En ce qui me concerne je vous ai donné la base pour démarrer, mais je serais bien incapable de vous montrer toutes les utilisations possibles de l'architecture modèle/vue de Qt. Nous ne pouvons pas tout voir, ce serait bien trop vaste.

A vous de voir, maintenant que vous commencez à connaître le principe, de quels outils vous avez besoin. Entraînez-vous avec les autres vues (arbre, tableau) et essayez d'en faire une utilisation plus poussée. Vous pouvez refaire un TP précédent et y intégrer un widget basé sur l'architecture modèle/vue pour vous entraîner.

Ce sera probablement difficile, mais bon, il faut bien des chapitres difficiles dans le cours sinon on va croire que c'est un site pour les débutants ici. 🤔

N'oubliez pas de lire la doc surtout, elle contient toutes les informations dont vous avez besoin !



Au fait, il existe des classes "simplifiées" des vues que l'on vient de voir. Ces classes s'appellent `QListWidget`, `QTreeWidget`, et `QTableWidget`. Elles ne nécessitent pas la création d'un modèle à part, mais sont du coup moins flexibles.

Utilisez-les lorsque vous avez une application très simple et que vous ne voulez pas manipuler l'architecture modèle/vue.

Communiquer en réseau avec son programme

Ah... Le réseau...

C'est un peu le fantasme de la plupart des nouveaux programmeurs : arriver à faire en sorte que son programme puisse communiquer à travers le réseau, que ce soit en local (entre 2 PC chez vous) ou sur internet.

Pourtant, c'est un sujet complexe parce que... il ne suffit pas seulement de savoir programmer en C++, il faut aussi beaucoup de connaissances théoriques sur le fonctionnement du réseau. Les couches d'abstraction, TCP/IP, UDP, Sockets... peut-être avez-vous entendu ces mots-là mais sauriez-vous vraiment les définir ?



En fait, pour que les choses soient claires, il faut savoir que je n'avais pas prévu de rédiger ce chapitre à la base. Tout d'abord parce que ce n'est plus vraiment du GUI (création de fenêtre), donc c'est un peu hors-sujet vis à vis des chapitres précédents. D'autre part, comme je vous l'ai dit, c'est un sujet complexe et il faudrait un tutoriel entier sur plusieurs chapitres pour bien vous expliquer la théorie sur les réseaux... chose que je ne peux pas faire sauf si vous me payez la greffe d'un troisième bras. 🤖

Cependant, vous êtes nombreux à m'avoir demandé de faire un chapitre traitant du réseau. Face à la demande, j'ai finalement accepté de faire une *exception*.

Exceptionnellement donc, nous n'allons pas vraiment parler que de GUI, nous allons aussi parler de réseau.

Seulement voilà, comme je vous l'ai dit, pour bien faire il faudrait un tutoriel complet que je n'ai ni le temps ni les moyens de rédiger. Du coup, j'ai finalement trouvé un compromis : on va faire une sorte de **chapitre-TP**. Il y aura de la théorie et de la pratique à la fois.

Nous ne verrons pas tout, nous nous concentrerons sur l'architecture réseau la plus classique (client / serveur). Cela vous donnera les bases pour comprendre un peu comment ça marche, et puis après il ne tiendra plus qu'à vous d'adapter ces exemples à vos programmes.



C'est un chapitre-TP ? Mais alors, quel est le sujet du TP ?

Le sujet du "chapitre-TP" sera la réalisation d'un **logiciel de Chat en réseau**. Vous pourrez aussi bien communiquer en réseau local (entre vos PC à la maison) qu'entre plusieurs PC via internet.

On y va ? 😊

On va devoir commencer par un petit cours théorique, absolument indispensable pour comprendre la suite de ce chapitre !

Comment communique-t-on en réseau ?

Voilà une bien bonne question !

A laquelle... je pourrais vous répondre par une encyclopédie en 12 volumes, et encore je n'aurais pas tout expliqué. 🤪

Nous allons donc voir les notions théoriques de base sur le réseau de façon *light* et ludique. A partir de là, nous pourrions voir comment on utilise ces connaissances en pratique avec Qt pour réaliser un Chat en réseau.

Pour nos exemples, nous allons imaginer 2 utilisateurs. Appelons-les... par exemple Patrice et Ludovic. Patrice et Ludovic ont chacun un ordinateur et ils voudraient communiquer entre eux.



Comment faire ? Comment communiquer, sachant qu'il y a des centaines, des milliers d'autres ordinateurs sur le réseau ? Et comment peuvent-ils se faire comprendre entre eux, faut-il qu'ils parlent le même langage ?

Pour que vous puissiez avoir 2 programmes qui communiquent entre eux via le réseau, il vous faut 3 choses :

1. Connaître l'**adresse IP** identifiant l'autre ordinateur.
2. Utiliser un **port** libre et ouvert.
3. Utiliser le même **protocole** de transmission des données.

Si tous ces éléments sont réunis, c'est bon. 😊

Voyons voir comment faire pour avoir tout ça...

1/ L'adresse IP : identification des machines sur le réseau

La première chose qui devrait vous préoccuper, c'est de savoir comment les ordinateurs font pour se reconnaître entre eux sur un réseau.

Comment fait Patrice pour envoyer un message à Ludovic et seulement à lui ?

Qu'est-ce qu'une IP ?

Il faut savoir que chaque ordinateur est identifié sur le réseau par ce qu'on appelle une adresse IP. C'est une série de nombres, par exemple :

85.215.27.118

Cette adresse représente un ordinateur. Lorsque vous connaissez l'adresse IP de la personne avec qui vous voulez communiquer, vous savez déjà au moins vers qui vous vous dirigez. 🗺️

Mais voilà, le problème, parce que sinon ça serait trop simple, c'est qu'un ordinateur peut avoir non pas une mais *plusieurs* IP. En général aujourd'hui, on peut considérer qu'un ordinateur a en moyenne 3 IP :

- **Une IP interne** : c'est le localhost, aussi appelé loopback. C'est une IP qui sert pour communiquer à soi-même. Pas très utile vu qu'on n'emprunte pas le réseau du coup, mais ça nous sera très pratique pour les tests vous verrez.
Exemple : 127.0.0.1
- **Une IP du réseau local** : si vous avez plusieurs ordinateurs en réseau chez vous, ils peuvent communiquer entre eux sans passer par internet grâce à ces IP. Elles sont propres au réseau de votre maison.
Exemple : 192.168.0.3
- **Une IP internet** : c'est l'IP utilisée pour communiquer avec tous les autres ordinateurs de la planète qui sont connectés à internet. 🌐
Exemple : 86.79.12.105

Patrice et Ludovic ont donc plusieurs IP, selon le niveau auquel on se place :



Si je vous raconte ça, c'est parce que nous aurons besoin d'utiliser l'une ou l'autre de ces IP en fonction de la distance qui sépare Patrice de Ludovic.

Si Patrice et Ludovic sont dans une même maison, reliés par un réseau local, nous utiliserons une IP du réseau local (en rouge sur mon schéma).

Si Patrice et Ludovic sont reliés par internet, nous utiliserons leur adresse internet (en vert).

Pour ce qui est de l'adresse localhost, elle peut nous servir pour "simuler" le fonctionnement du réseau. Si Patrice envoie un message à 127.0.0.1, celui-ci va immédiatement lui revenir. Cela peut nous être donc utile si on ne veut pas déranger notre ami Ludovic toutes les 5 minutes pour tester la dernière version de notre programme. 🤖

Retrouver son adresse IP



Comment je connais mon IP ? Ou plutôt mes IP ?

Et comment je sais laquelle correspond au réseau local, et laquelle correspond à celle d'internet ?

La méthode dépend de l'IP que vous recherchez.

- **Pour l'IP interne** : pas besoin d'aller chercher plus loin, à coup sûr c'est 127.0.0.1 (ou son équivalent texte "localhost").
- **Pour l'IP locale** : pour la retrouver tout dépend de votre système d'exploitation.
 - **Sous Windows**, ouvrez une invite de commande (par exemple celui que vous utilisez avec Qt pour compiler) et tapez `ipconfig`
Il est possible que vous ayez plusieurs réponses, en fonction des moyens de connexion disponibles (câble ethernet, wifi...). En tout cas, l'une des IP que l'on vous donne est la bonne (à la ligne "Adresse IPv4").
 - **Sous Linux ou Mac OS**, c'est le même principe dans une console mais pas la même commande : `ifconfig`
L'adresse est en général de la forme "192.168.XXX.XXX", mais cela peut être parfois différent.
- **Pour l'IP internet** : le plus simple est probablement d'aller sur un site web qui est capable de vous la donner, comme par exemple www.whatismyip.com !

Maintenant que vous connaissez l'adresse IP de votre interlocuteur, alors vous allez pouvoir communiquer avec lui... ou presque. Le problème, c'est qu'il y a plusieurs portes d'entrée sur chaque ordinateur. C'est ce qu'on appelle **les ports**.

2/ Les ports : différents moyens d'accès à un même ordinateur

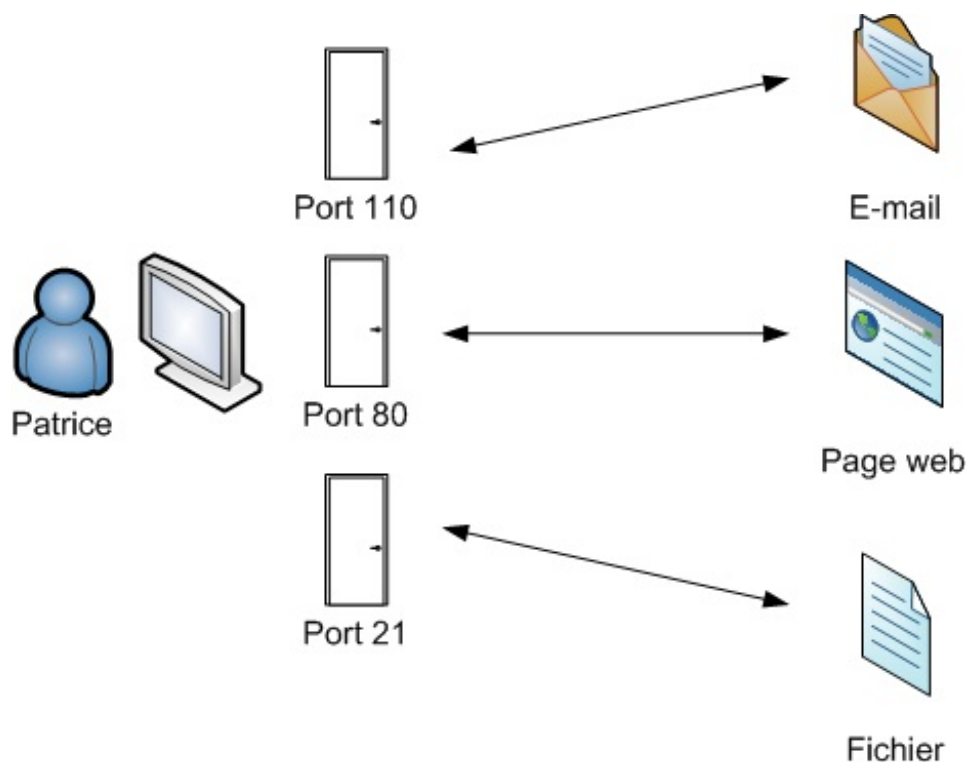
Un ordinateur connecté à un réseau reçoit beaucoup de messages en même temps.

Par exemple, si vous allez sur un site web en même temps que vous récupérez vos mails, des données différentes vont vous arriver simultanément.

Pour ne pas confondre ces données et organiser tout ce bazar, on a inventé le concept de port. Un port est un nombre compris entre 1 et 65 536. Voici quelques ports célèbres :

- **21** : utilisé par les logiciels FTP pour envoyer et recevoir des fichiers.
- **80** : utilisé pour naviguer sur le web par votre navigateur (par exemple Firefox, ou plutôt zNavigo 🤖).
- **110** : utilisé pour la réception de mails.

Imaginez que ces ports sont autant de portes d'entrée à votre ordinateur :



Si on veut faire un programme qui communique avec Ludovic, il va falloir choisir un port qui ne soit pas déjà utilisé par un autre programme.

La plupart des ports dont les numéros sont inférieurs à 1 024 sont déjà réservés par votre machine. Nous ferons donc en sorte de préférence dans notre programme d'utiliser un numéro de port compris entre 1 024 et 65 536.



Pour éviter que n'importe quel programme puisse communiquer sur le réseau et accéder à une machine sans autorisation, on a inventé les **firewalls** (pare-feu). Leur rôle est de bloquer tous les ports d'une machine, et d'en autoriser seulement certains qui sont considérés comme "sûrs" (comme les ports 21, 80, 110...).

Il faudra bien vérifier la configuration de votre firewall si vous en avez un (il y en a un activé sous Windows par défaut depuis Windows XP SP2), car celui-ci pourrait tout simplement bloquer les communications de notre programme !

3/ Le protocole : transmettre des données avec le même "langage"

Bon, nous savons désormais 2 choses :

- Chaque ordinateur est identifié par une **adresse IP**.
- On peut accéder à une IP via des milliers de **ports** différents.

L'IP, vous savez la retrouver. Le port, il faudra en choisir un qui soit libre (nous verrons comment en pratique plus tard). Vous êtes donc maintenant en mesure d'établir une connexion avec un ordinateur distant, car vous avez les 2 éléments nécessaires : une IP et un port.

Il reste maintenant à envoyer des données à l'ordinateur distant pour que les 2 programmes puissent "parler" entre eux. Et ça mine de rien, ce n'est pas simple. En effet, il faut que les 2 programmes parlent la même langue, le même **protocole**. Il faut qu'ils communiquent de la même façon.



Définition : un protocole est un ensemble de règles qui permettent à 2 ordinateurs de communiquer. Il faut impérativement que les 2 ordinateurs parlent le même protocole pour que l'échange de données puisse fonctionner.

Exemple de la vie courante : vous dites "Bonjour" lorsque vous commencez à parler à quelqu'un, et "Au revoir" lorsque vous partez. Eh bien pour les ordinateurs c'est pareil !

Les différents niveaux des protocoles de communication

Il existe des centaines de protocoles de communication différents. Ceux-ci peuvent être très simples comme très complexes, selon si vous discutez à un "haut niveau" ou à un "bas niveau". On peut donc les ranger dans 2 catégories :

- **Protocoles de haut niveau** : par exemple le protocole FTP, qui utilise le port 21 pour envoyer et recevoir des fichiers, est un système d'échange de données de haut niveau. Son mode de fonctionnement est déjà écrit et documenté. Il est donc assez facile à utiliser, mais on ne peut pas lui rajouter des possibilités.
- **Protocoles de bas niveau** : par exemple le protocole TCP. Il est utilisé par les programmes pour lesquels aucun protocole de haut niveau ne convient. Vous devrez manipuler les données qui transitent sur le réseau octet par octet. C'est plus difficile, mais vous pouvez faire tout ce que vous voulez.

Protocoles haut niveau

Tous prêts et faciles à utiliser

HTTP (port 80)
FTP (port 21)
POP3 (port 110)

Protocoles bas niveau

Difficiles à utiliser
Mode de communication à
définir soi-même

TCP (port à choisir)
UDP (port à choisir)



Pour ceux qui veulent aller plus loin, renseignez-vous sur le [modèle OSI](#). C'est un modèle d'organisation des données sur le réseau qui vous explique les différents niveaux de communication (on parle de "couches").

Ici j'ai beaucoup (énormément) simplifié le schéma, mais sinon on ne s'en sortait pas en un seul chapitre. 😊

Les protocoles de haut niveau utilisent des ports bien connus et déjà définis.

Les protocoles de bas niveau peuvent emprunter n'importe quel port, sont beaucoup plus flexibles, mais le problème c'est qu'il faut définir tout leur fonctionnement.



En fait, tous les protocoles de haut niveau utilisent des protocoles de bas niveau pour leur fonctionnement interne. Les protocoles de bas niveau sont "la base", on les utilise pour construire des protocoles de plus haut niveau.

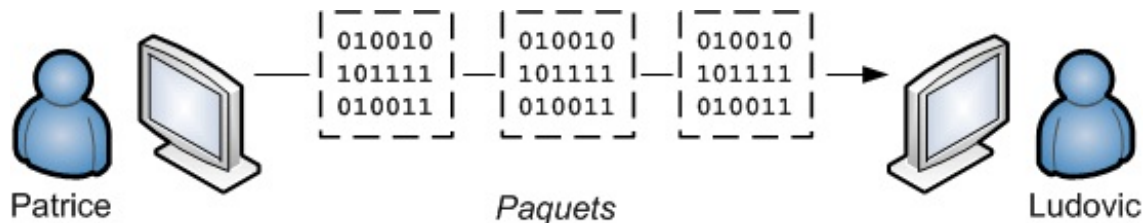
Nous n'allons pas créer un logiciel de mails, ni un client FTP. Nous allons inventer notre propre technique de discussion pour notre programme, notre propre protocole basé sur un protocole de bas niveau... Nous allons donc travailler à bas niveau.

Mauvaise nouvelle : ça va être plus difficile. 😞

Bonne nouvelle : ça va être intéressant techniquement.

Les protocoles de bas niveau TCP et UDP

Il faut savoir que les données s'envoient sur le réseau par petits bouts. On parle de **paquets**, qui peuvent être chacun découpés en sous-paquets :



Par exemple, imaginons que Patrice envoie à Ludovic le message : "Salut Ludovic, comment ça va ?". Le message ne sera peut-être pas envoyé d'un seul coup, il sera probablement découpé en plus petits paquets. Par exemple, on peut imaginer qu'il y aura 4 sous-paquets (j'invente, car le découpage sera peut-être différent) :

1. **Sous-paquet 1** : "Salut Ludov"
2. **Sous-paquet 2** : "ic, co"
3. **Sous-paquet 3** : "mment ça v"
4. **Sous-paquet 4** : "a ?"



Ce n'est pas vous qui gérez le découpage en sous-paquets, c'est le protocole de bas niveau qui s'en occupe. Il est donc impossible de connaître à l'avance la taille des paquets ou même leur nombre. Il est cependant important de savoir que ça fonctionne comme ça pour la suite.

On peut envoyer ces paquets de plusieurs façons différentes, tout dépend du protocole de bas niveau que l'on utilise :

- **Protocole TCP** : le plus classique. Il nécessite d'établir une connexion au préalable entre les ordinateurs. Il y a un système de contrôle qui permet de demander à renvoyer un paquet au cas où l'un d'entre eux se serait perdu sur le réseau (ça arrive 😞). Par conséquent, avec TCP on est sûr que tous les paquets arrivent à destination, et dans le bon ordre. En contrepartie de ces contrôles sécurisants, l'envoi des données est plus lent qu'avec UDP.
- **Protocole UDP** : il ne nécessite pas d'établir de connexion au préalable et il est très rapide. En revanche, il n'y a aucun contrôle ce qui fait qu'un paquet de données peut très bien se perdre sans qu'on en soit informé, ou les paquets peuvent arriver dans le désordre !

Il va falloir choisir l'un de ces 2 protocoles.

Pour moi, le choix est tout fait : ce sera TCP. En effet, nous allons réaliser un Chat et nous ne pouvons pas nous permettre que des messages (ou des bouts de messages) n'arrivent pas à destination, sinon la conversation pourrait devenir difficile à suivre et on risquerait de recevoir des messages comme : "Salut Ludovmment ça va ?" 😞



Mais alors, du coup tout le monde utilise TCP pour être sûr que le paquet arrive à destination non ? Qui peut bien être assez fou pour utiliser UDP ?

Certaines applications complexes qui utilisent beaucoup le réseau peuvent être amenées à utiliser UDP. Je pense par exemple aux jeux vidéo.

Prenez un jeu de stratégie comme Starcraft, ou un FPS comme Quake par exemple : il peut y avoir des dizaines d'unités qui se déplacent sur la carte en même temps. Il faut en continu envoyer la nouvelle position des unités qui se déplacent à tous les ordinateurs de la partie. On a donc besoin d'un protocole rapide comme UDP, et si un paquet se perd ce n'est pas grave : vu que

la position des joueurs est rafraîchie plusieurs fois par seconde, ça ne se verra pas.

L'architecture du projet de Chat avec Qt

Nous venons de voir quelques petites notions théoriques sur le réseau, mais il va encore falloir préciser quelle est l'architecture réseau de notre programme de Chat.



Une architecture réseau ? Qu'est-ce que c'est que ça ? 🤔

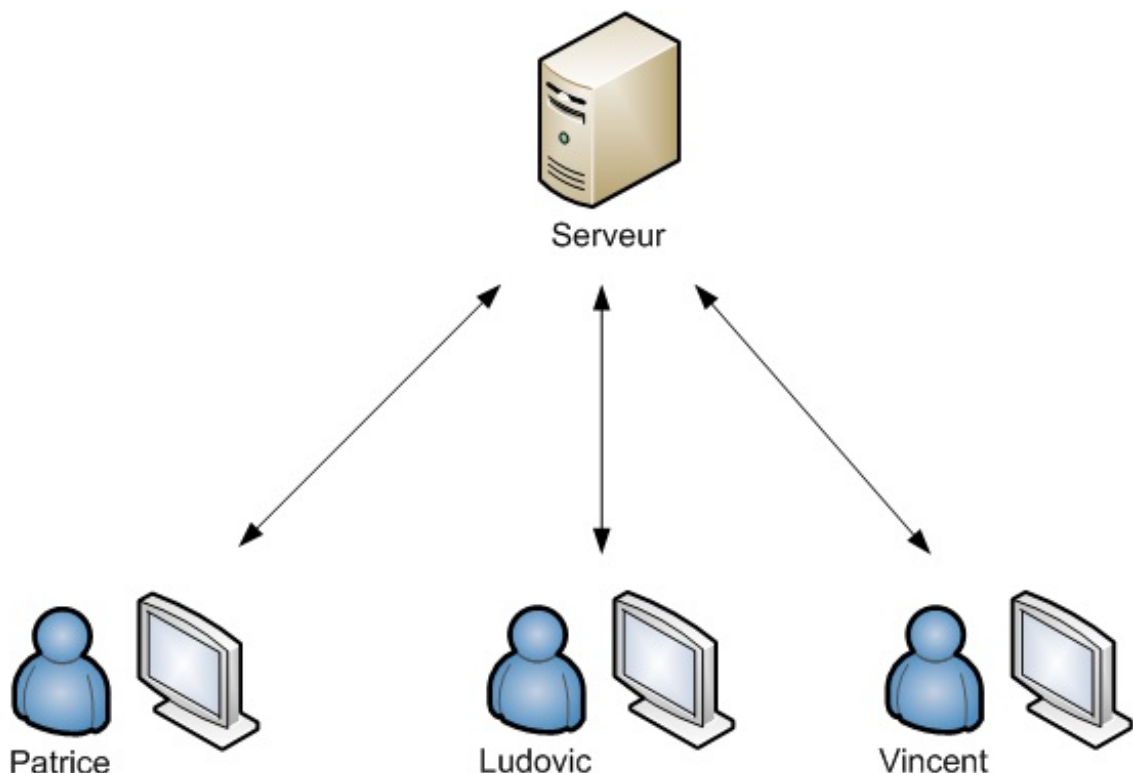
Jusqu'ici, nous avons supposé un cas très simple : il n'y avait que 2 ordinateurs (celui de Patrice et celui de Ludovic). Le problème, c'est que notre programme de Chat doit permettre à plus de 2 personnes de discuter en même temps. Imaginons qu'une troisième personne appelée Vincent arrive sur le Chat. Vous le placez où sur le schéma ? Au milieu entre les 2 autres compères ?



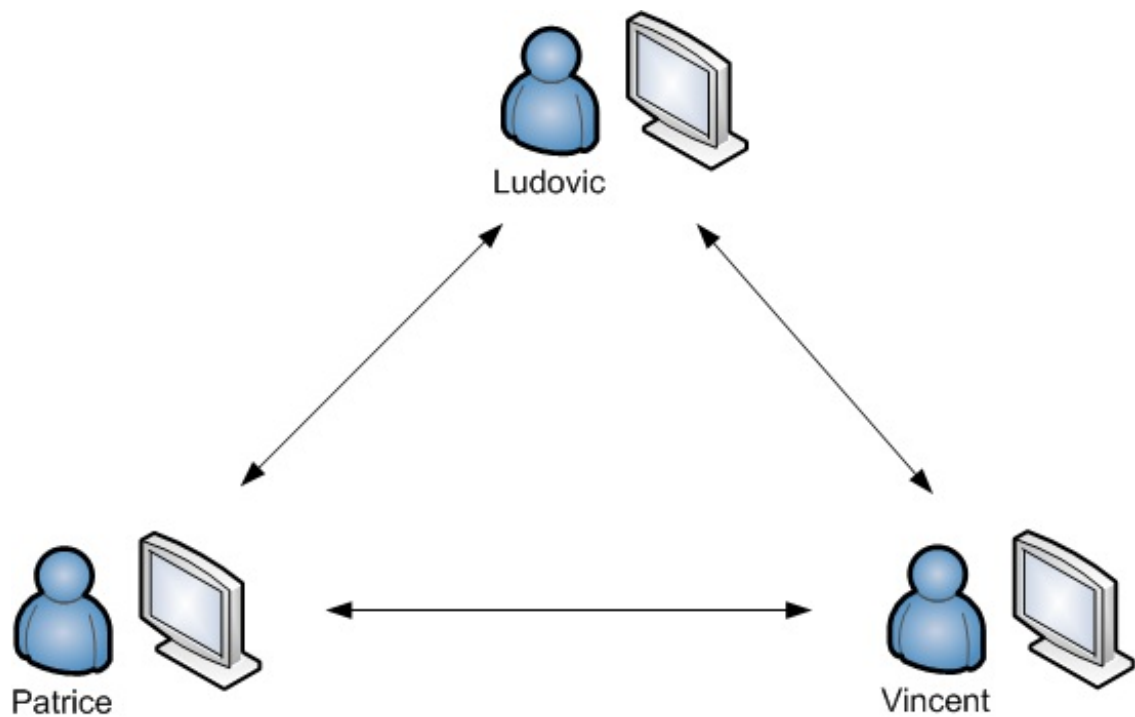
Les architectures réseau

Pour faire simple, on a 2 architectures possibles pour résoudre le problème :

- **Une architecture client / serveur** : c'est l'architecture réseau la plus classique et la plus simple à mettre en oeuvre. Les machines des utilisateurs (Patrice, Ludovic, Vincent...) sont appelées des "*clients*". En plus de ces machines, on utilise un autre ordinateur (appelé "*serveur*") qui va se charger de répartir les communications entre les clients.



- **Une architecture Peer-To-Peer (P2P)** : ce mode plus complexe est dit décentralisé, car il n'y a pas de serveur. Chaque client peut communiquer directement avec un autre client. C'est plus direct, ça évite d'encombrer un serveur, mais c'est plus délicat à mettre en place.



Nous, nous allons utiliser une **architecture client / serveur**, la plus simple.

Il va en fait falloir faire non pas un mais deux projets :

- Un projet "**serveur**" : pour créer le programme qui va répartir les messages entre les clients.
- Un projet "**client**" : pour chaque client qui participera au Chat.



Vous n'êtes pas obligés d'utiliser une machine spécialement pour faire serveur. L'une des machines des clients peut aussi faire office de serveur. Il suffira de faire tourner un programme "serveur" en même temps qu'un programme "client", c'est tout à fait possible.

En pratique donc, une seule personne lancera le programme "serveur" et le programme "client" à la fois, et toutes les autres lanceront uniquement le programme "client".

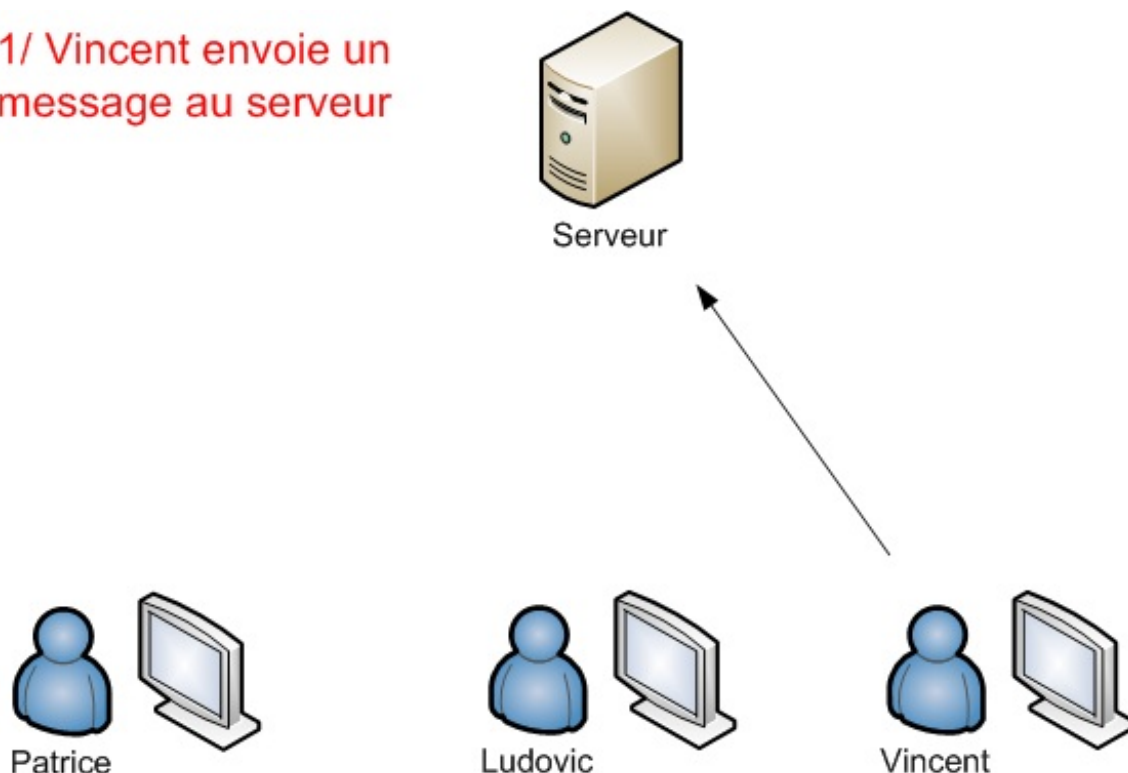
Principe de fonctionnement du Chat

Le principe du Chat est simple : une personne écrit un message, et tout le monde reçoit ce message sur son écran.

Les choses se passent en 2 temps :

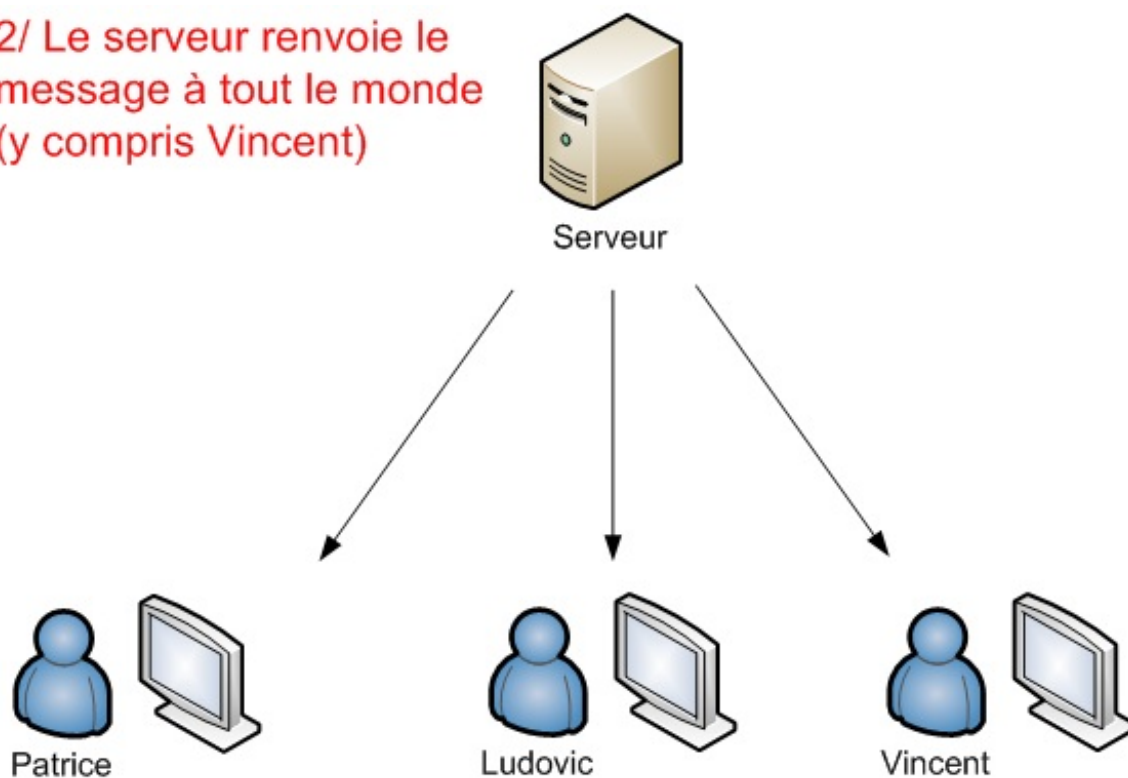
- Un client envoie un message au serveur.

1/ Vincent envoie un message au serveur



- Le serveur renvoie ce message à tous les clients pour qu'il s'affiche sur leur fenêtre.

2/ Le serveur renvoie le message à tout le monde (y compris Vincent)



Pourquoi le serveur renverrait-il le message à Vincent, vu que c'est lui qui l'a envoyé ?

On peut gérer les choses de plusieurs manières. On pourrait s'arranger pour que le serveur n'envoie pas le message à Vincent pour éviter un trafic réseau inutile, mais cela compliquerait un petit peu le programme.

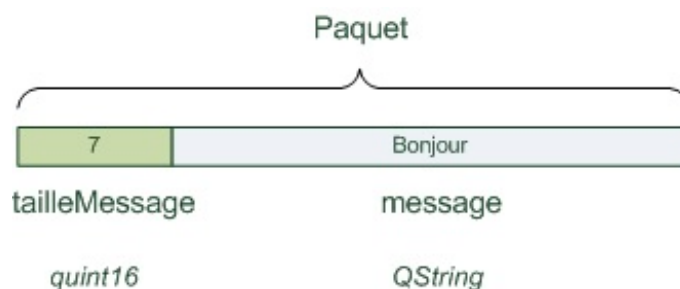
Il est plus simple de faire en sorte que le serveur renvoie le message à tout le monde sans distinction. Vincent verra donc son message s'afficher sur son écran de discussion uniquement quand le serveur l'aura reçu et le lui aura renvoyé. Cela permet de

vérifier en outre que la communication sur le réseau fonctionne correctement.

Structure des paquets

Les messages qui circuleront sur le réseau seront placés dans des **paquets**. C'est à nous de définir la structure des paquets que l'on veut envoyer.

Par exemple, quand Vincent va envoyer un message, un paquet va être créé avant d'être envoyé sur le réseau. Voici la structure de paquet que je propose pour notre programme de Chat :



Le paquet est constitué de 2 parties :

- **tailleMessage** : un nombre entier qui sert à indiquer la taille du message qui suit. Cela permet au serveur de connaître la taille totale du message envoyé, pour qu'il puisse savoir quand il a reçu le message en entier.



Ce nombre ne sera pas de type *int* comme on aurait pu s'y attendre mais de type *quint16*. En effet, le type *int* peut avoir une taille différente selon les machines sur le réseau (un *int* peut prendre 16 bits de mémoire sur une machine, et 8 bits sur une autre).

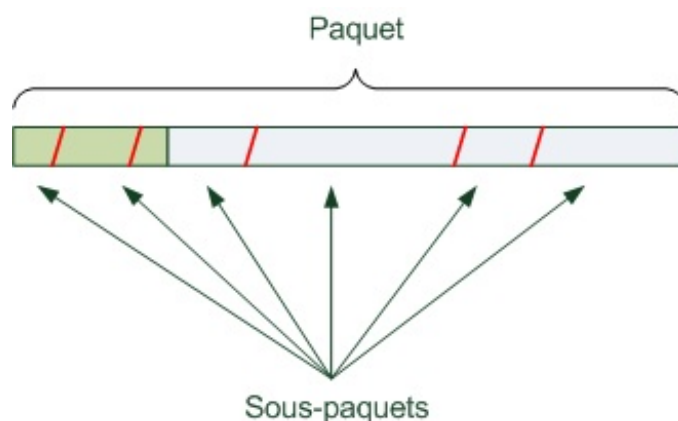
Pour résoudre le problème, on utilise un type spécial de Qt, le *quint16*, qui correspond à un nombre entier prenant 16 bits de mémoire quelle que soit la machine (quand je vous avais dit que c'était bas niveau 😊). *quint16* signifie : "Qt Unsigned Int 16", soit "Entier non signé codé sur 16 bits".

- **message** : c'est le message envoyé par le client. Ce message sera de type *QString* (ça c'est simple, vous connaissez !).



Pourquoi envoie-t-on la taille du message en premier ? On ne pourrait pas envoyer le message tout court ?

Il faut savoir que le protocole TCP va découper le paquet en sous-paquets avant de l'envoyer sur le réseau. Il n'enverra peut-être pas tout d'un coup. Par exemple, notre paquet pourrait être découpé comme ceci :



On n'a aucun contrôle sur la taille de ces sous-paquets, et il n'y a aucun moyen de savoir à l'avance comment ça va être découpé. Le problème, c'est que le serveur va recevoir ces paquets petit à petit, et non pas tout d'un coup. Il ne *peut pas savoir* quand la totalité du message a été reçue.



Le protocole TCP ne permet pas de contrôler la taille des sous-paquets ni leur nombre, par contre il s'arrange pour que les paquets arrivent à destination dans le bon ordre (ce qui est pratique, parce que sinon ça aurait été encore plus compliqué à remettre en ordre 😊).

Pour information, le protocole UDP, qui est plus rapide, ne fait aucun contrôle sur l'ordre des paquets envoyés !

Bon, il faut qu'on arrive à savoir *quand* on a reçu le message en entier, et donc quand ce n'est plus la peine d'attendre de nouveaux sous-paquets.

Pour résoudre ce problème, on envoie la taille du message dans un premier temps. Lorsque la taille du message a été reçue, on va attendre que le message soit au complet. On se base sur `tailleMessage` pour savoir combien d'octets il nous reste à recevoir.

Lorsqu'on a récupéré tous les octets restants du paquet, on sait que le paquet est au complet, et cela veut dire qu'on a donc reçu le message entier.

Bon, c'est pas simple, mais je vous avais prévenu hein ! 😊

Réalisation du serveur

Comme je vous l'ai dit, nous allons devoir réaliser 2 projets :

- Un projet "client"
- Un projet "serveur"

Nous commençons par le serveur.

Création du projet

Créez un nouveau projet constitué de 3 fichiers :

- `main.cpp`
- `FenServeur.cpp`
- `FenServeur.h`

Editez le fichier `.pro` pour demander à Qt de rajouter la gestion du réseau :

Code : Autre

```
#####
# Automatically generated by qmake (2.01a) mer. 25. juin 14:54:55 2008
#####

TEMPLATE = app
QT += network
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

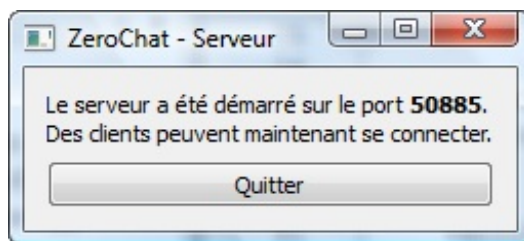
# Input
HEADERS += FenServeur.h
SOURCES += FenServeur.cpp main.cpp
```

Avec `QT += network`, Qt sait que le projet va utiliser le réseau et peut préparer un makefile approprié.

La fenêtre du serveur

Le serveur est une application qui tourne en tâche de fond. Normalement, rien ne nous oblige à créer une fenêtre pour ce projet, mais on va quand même en faire une pour que l'utilisateur puisse arrêter le serveur en fermant la fenêtre.

Notre fenêtre sera toute simple, elle affichera le texte "*Le serveur a été lancé sur le port XXXX*" et un bouton "*Quitter*".



Construire la fenêtre sera donc très simple, la vraie difficulté sera de faire toute la gestion du réseau derrière.

main.cpp

Les main sont toujours très simples et classiques avec Qt :

Code : C++

```
#include <QApplication>
#include "FenServeur.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenServeur fenetre;
    fenetre.show();

    return app.exec();
}
```

FenServeur.h

Voici maintenant le header de la fenêtre du serveur :

Code : C++

```
#ifndef HEADER_FENSERVEUR
#define HEADER_FENSERVEUR

#include <QtGui>
#include <QtNetwork>

class FenServeur : public QWidget
{
    Q_OBJECT

public:
    FenServeur();
    void envoyerATous(const QString &message);

private slots:
    void nouvelleConnexion();
    void donneesRecues();
}
```

```

        void deconnexionClient();

    private:
        QLabel *etatServeur;
        QPushButton *boutonQuitter;

        QTcpServer *serveur;
        QList<QTcpSocket *> clients;
        quint16 tailleMessage;
};

#endif

```

Notre fenêtre hérite de QWidget, ce qui nous permet de créer une fenêtre simple. Elle est constituée comme vous le voyez d'un QLabel et d'un QPushButton comme prévu.

En plus de ça, j'ai rajouté d'autres attributs spécifiques à la gestion du réseau :

- `QTcpServer *serveur` : c'est l'objet qui représente le serveur sur le réseau.
- `QList<QTcpSocket *> clients` : c'est un tableau qui contient la liste des clients connectés. On aurait pu utiliser un tableau classique, mais on va passer par une QList, un tableau de taille dynamique. En effet, on ne connaît pas à l'avance le nombre de clients qui se connecteront. Chaque QTcpSocket de ce tableau représentera une connexion à un client.



Je ne vais pas m'étendre dans ce chapitre sur les [QList](#). Considérez juste que c'est une classe qui permet de gérer un tableau de taille variable, ce qui est très pratique quand on ne sait pas comme ici le nombre de clients qui vont se connecter.

Pour plus d'infos pour apprendre à vous en servir, lisez la doc. 😊

- `quint16 tailleMessage` : ce quint16 sera utilisé dans le code pour se "souvenir" de la taille du message que le serveur est en train de recevoir. Nous en avons déjà parlé et nous en reparlerons plus loin.

Voilà, à part ces attributs, on note que la classe est constituée de plusieurs méthodes (dont des slots) :

- **Le constructeur** : il initialise les widgets sur la fenêtre et initialise aussi le serveur (QTcpServer) pour qu'il démarre.
- **envoyerATous()** : une méthode à nous qui se charge d'envoyer à tous les clients connectés le message passé en paramètre.
- **Slot nouvelleConnexion()** : appelé lorsqu'un nouveau client se connecte.
- **Slot donneesRecues()** : appelé lorsque le serveur reçoit des données. Attention, c'est là que c'est délicat, car ce slot est appelé à chaque sous-paquet reçu. Il faudra "attendre" d'avoir reçu le nombre d'octets indiqués dans `tailleMessage` avant de pouvoir considérer qu'on a reçu le message entier.
- **Slot deconnexionClient()** : appelé lorsqu'un client se déconnecte.

Implémentons ces méthodes en coeur, dans la joie et la bonne humeur ! 😊

FenServeur.cpp

Le constructeur

Le constructeur se charge de placer les widgets sur la fenêtre et de faire démarrer le serveur via le QTcpServer :

Code : C++

```

FenServeur::FenServeur()
{
    // Création et disposition des widgets de la fenêtre
}

```

```

    etatServeur = new QLabel;
    boutonQuitter = new QPushButton(tr("Quitter"));
    connect(boutonQuitter, SIGNAL(clicked()), qApp, SLOT(quit()));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(etatServeur);
    layout->addWidget(boutonQuitter);
    setLayout(layout);

    setWindowTitle(tr("ZeroChat - Serveur"));

    // Gestion du serveur
    serveur = new QTcpServer(this);
    if (!serveur->listen(QHostAddress::Any, 50885)) // Démarrage du
    serveur sur toutes les IP disponibles et sur le port 50885
    {
        // Si le serveur n'a pas été démarré correctement
        etatServeur->setText(tr("Le serveur n'a pas pu être démarré.
Raison :<br />") + serveur->errorString());
    }
    else
    {
        // Si le serveur a été démarré correctement
        etatServeur->setText(tr("Le serveur a été démarré sur le
port <strong>") + QString::number(serveur->serverPort()) +
tr("</strong>.<br />Des clients peuvent maintenant se connecter."));
        connect(serveur, SIGNAL(newConnection()), this,
SLOT(nouvelleConnexion()));
    }

    tailleMessage = 0;
}

```

J'ai fait en sorte de bien commenter mes codes sources pour vous aider du mieux possible à comprendre ce qui se passe. Vous voyez bien une première étape où on dispose les widgets sur la fenêtre (classique, rien de nouveau) et une seconde étape où on démarre le serveur.

Quelques précisions sur la seconde étape, la plus intéressante pour ce chapitre. On crée un nouvel objet de type `QTcpServer` dans un premier temps (ligne 16). On lui passe en paramètre `this`, un pointeur vers la fenêtre, pour faire en sorte que la fenêtre soit le parent du `QTcpServer`. Cela permet de faire en sorte que le serveur soit automatiquement détruit lorsqu'on quitte la fenêtre.

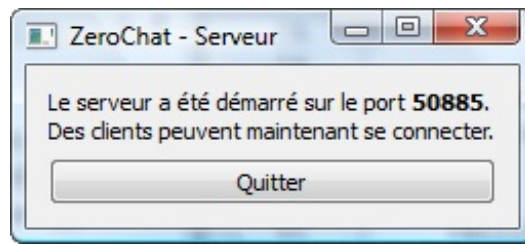
Ensuite, on essaie de démarrer le serveur grâce à `serveur->listen(QHostAddress::Any, 50885)`. Il y a 2 paramètres :

- **L'IP** : c'est l'IP sur laquelle le serveur "écoute" si de nouveaux clients arrivent. Comme je vous l'avais dit, un ordinateur peut avoir plusieurs IP : une IP interne (127.0.0.1), une IP pour le réseau local, une IP sur internet, etc. La mention `QHostAddress::Any` autorise toutes les connexions : internes (clients connectés sur la même machine), locales (clients connectés sur le même réseau local) et externes (clients connectés via internet).
- **Le port** : c'est le numéro du port sur lequel on souhaite lancer le serveur. J'ai choisi un numéro au hasard, compris entre 1 024 et 65 536. J'aurais aussi pu omettre ce paramètre, dans ce cas le serveur aurait choisi un port libre au hasard. N'hésitez pas à changer la valeur si le port n'est pas libre chez vous.

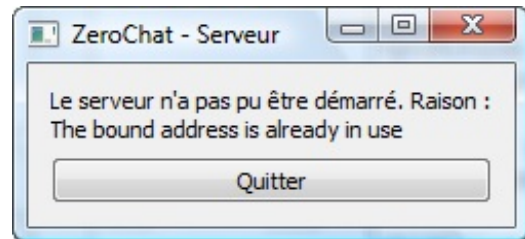
La méthode `listen()` renvoie un booléen : vrai si le serveur a bien pu se lancer, faux s'il y a eu un problème. On affiche un message en conséquence sur la fenêtre du serveur.

Si le démarrage du serveur a fonctionné, on connecte le signal `newConnection()` vers notre slot personnalisé `nouvelleConnexion()` pour traiter l'arrivée d'un nouveau client sur le serveur.

Si tout va bien, la fenêtre suivante devrait donc s'ouvrir :



S'il y a une erreur, vous aurez un message d'erreur adapté. Par exemple, essayez de lancer une seconde fois le serveur alors qu'un autre serveur tourne déjà :



Ici, comme le port 50885 est déjà utilisé par un 1er serveur, notre 2nd serveur n'a pas le droit de démarrer sur ce port. D'où l'erreur.



Slot nouvelleConnexion()

Ce slot est appelé dès qu'un nouveau client se connecte au serveur :

Code : C++

```
void FenServeur::nouvelleConnexion()
{
    envoyerATous(tr("<em>Un nouveau client vient de se
connecter</em>"));

    QTcpSocket *nouveauClient = serveur->nextPendingConnection();
    clients << nouveauClient;

    connect(nouveauClient, SIGNAL(readyRead()), this,
    SLOT(donneesRecues()));
    connect(nouveauClient, SIGNAL(disconnected()), this,
    SLOT(deconnexionClient()));
}
```

On envoie à tous les clients déjà connectés un message comme quoi un nouveau client vient de se connecter. On verra le contenu de la méthode `envoyerATous()` un peu plus loin.

Chaque client est représenté par un `QTcpSocket`. Pour récupérer la socket correspondant au nouveau client qui vient de se connecter, on appelle la méthode `nextPendingConnection()` du `QTcpServer`. Cette méthode retourne la `QTcpSocket` du nouveau client.

Comme je vous l'ai dit, on conserve la liste des clients connectés dans un tableau, appelé *clients*.

Ce tableau est géré par la classe `QList` qui est très simple d'utilisation. On ajoute le nouveau client à la fin du tableau très facilement, comme ceci :

Code : C++

```
clients << nouveauClient;
```

(vive la surcharge de l'opérateur << 😊)

On connecte ensuite les signaux que peut envoyer le client à des slots. On va gérer 2 signaux :

- **readyRead()** : signale que le client a envoyé des données. Ce signal est émis pour chaque sous-paquet reçu. Lorsqu'un client enverra un message, ce signal pourra donc être émis plusieurs fois jusqu'à ce que tous les sous-paquets soient arrivés.
C'est notre slot personnalisé `donneesRecues()` (qui sera coton à écrire 😊) qui traitera les sous-paquets.
- **disconnected()** : signale que le client s'est déconnecté. Notre slot se chargera d'informer les autres clients de son départ et de supprimer la `QTcpSocket` correspondante dans la liste des clients connectés.

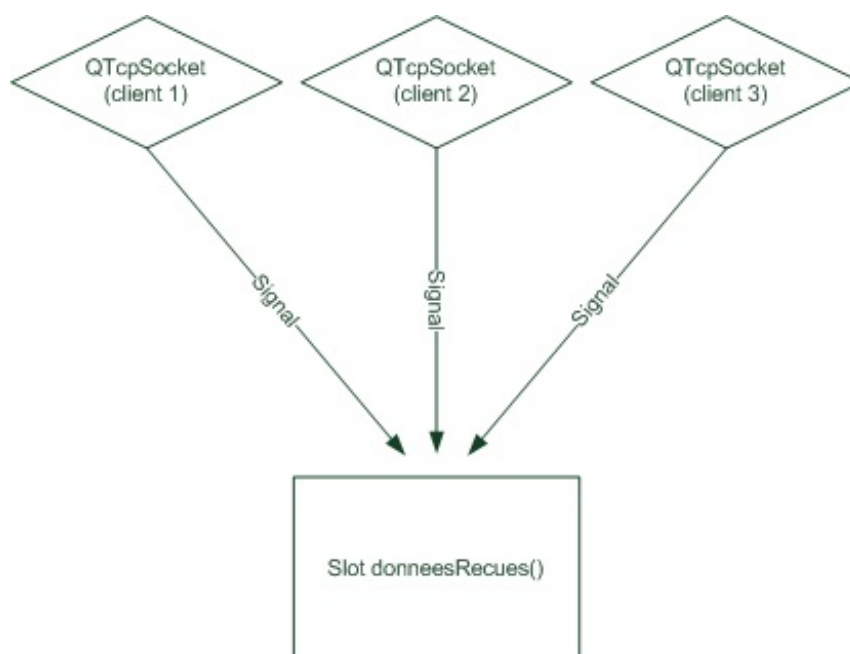
Slot `donneesRecues()`

Voilà sans aucun doute LE point le plus délicat de ce chapitre. C'est un slot qui va être appelé à chaque fois qu'on reçoit un sous-paquet d'un des clients.

On a au moins 2 problèmes pas évidents à résoudre :

- Comme on va recevoir plusieurs sous-paquets, il va falloir "attendre" d'avoir tout reçu avant de pouvoir dire qu'on a reçu le message en entier.
- C'est le même slot qui est appelé quel que soit le client qui a envoyé un message. Du coup, comment savoir quel est le client à l'origine du message pour récupérer les données ?

Il faut utiliser l'objet `QTcpSocket` du client pour récupérer les sous-paquets qui ont transité par le réseau. Le problème, c'est qu'on a connecté les signaux de tous les clients à un même slot :



Comment le slot sait-il dans quelle `QTcpSocket` lire les données ?

Vous ne pouviez pas trop le deviner, et à vrai dire je ne savais pas moi-même comment faire avant d'écrire ce chapitre 😊.

Il se trouve que j'ai découvert qu'on pouvait appeler la méthode `sender()` de `QObject` dans le slot pour retrouver un pointeur vers l'objet à l'origine du message. Très pratique ! 😊

Nouveau problème : cette méthode renvoie systématiquement un `QObject` (classe générique de Qt) car elle ne sait pas à l'avance de quel type sera l'objet. Notre objet `QTcpSocket` sera donc représenté par un `QObject`.

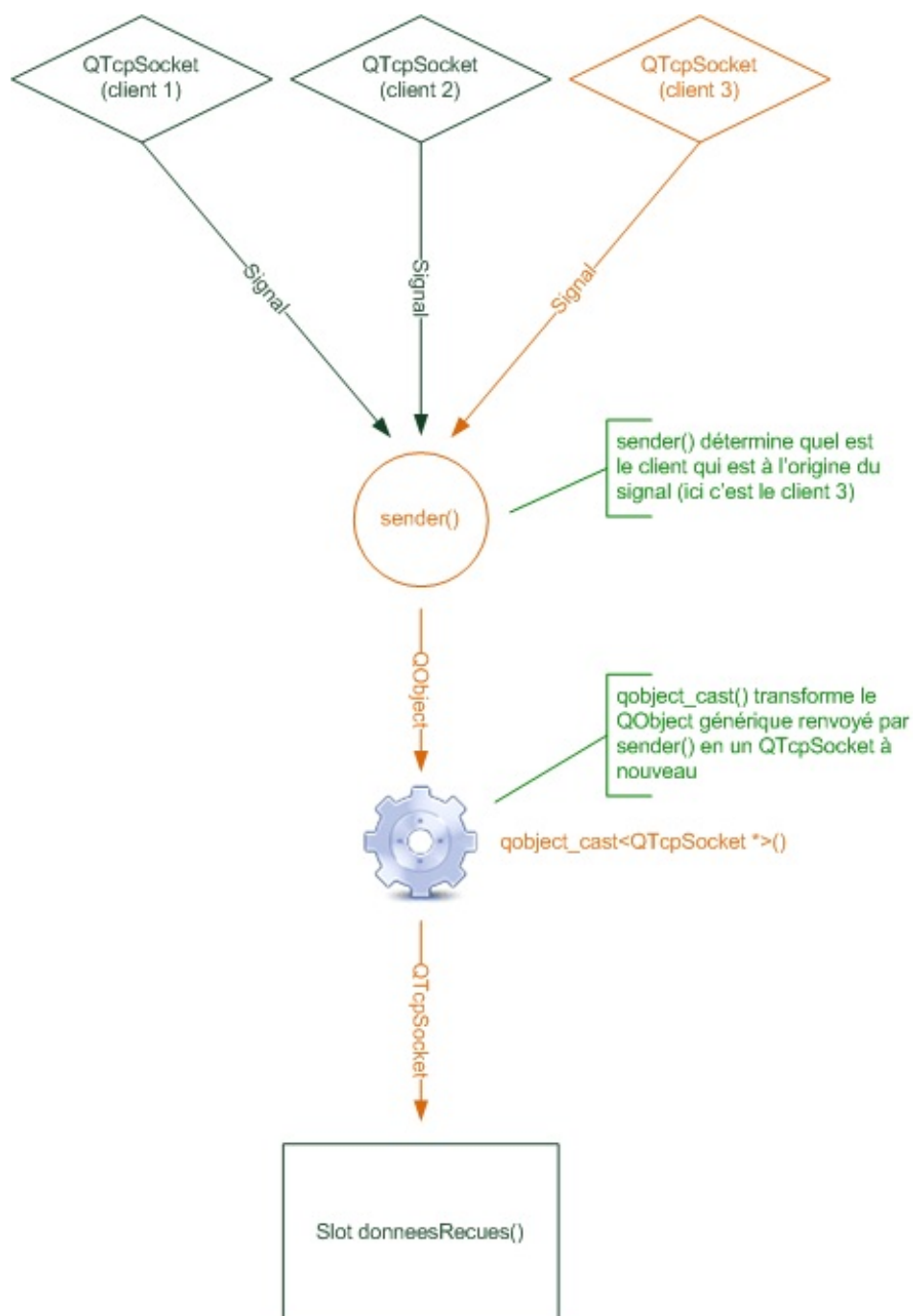
Pour le transformer à nouveau en `QTcpSocket`, il faudra forcer sa conversion à l'aide de la méthode `qobject_cast()`.

En résumé, pour obtenir un pointeur vers la bonne `QTcpSocket` à l'origine du signal, il faudra écrire :

Code : C++

```
QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
```

Ce qui, schématiquement, revient à faire ceci :



1. On utilise `sender()` pour déterminer l'objet à l'origine du signal.
2. Comme `sender()` renvoie systématiquement un `QObject`, il faut le transformer à nouveau en `QTcpSocket`. Pour cela, on passe l'objet en paramètre à la méthode `qobject_cast()`, on indiquant entre les chevrons le type de retour que l'on souhaite obtenir : `<QTcpSocket *>`.



La méthode `qobject_cast()` est similaire au `dynamic_cast()` de la bibliothèque standard du C++. Son rôle est de forcer la transformation d'un objet d'un type vers un autre.

Il se peut que le `qobject_cast()` n'ait pas fonctionné (par exemple parce que l'objet n'était pas de type `QTcpSocket` contrairement à ce qu'on attendait). Dans ce cas, il renvoie 0. Il faut que l'on teste si le `qobject_cast()` a fonctionné avant d'aller plus loin. On va faire un `return` qui va arrêter la méthode s'il y a eu un problème :

Code : C++

```
QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());  
if (socket == 0) // Si par hasard on n'a pas trouvé le client à  
l'origine du signal, on arrête la méthode  
    return;
```

On peut ensuite travailler à récupérer les données. On commence par créer un flux de données pour lire ce que contient la socket :

Code : C++

```
QDataStream in(socket);
```

Notre objet "in" va nous permettre de lire le contenu du sous-paquet que vient de recevoir la socket du client.

C'est maintenant que l'on va utiliser l'entier `tailleMessage` défini en tant qu'attribut de la classe. Si lors de l'appel au slot ce `tailleMessage` vaut 0, cela signifie qu'on est en train de recevoir le début d'un nouveau message.

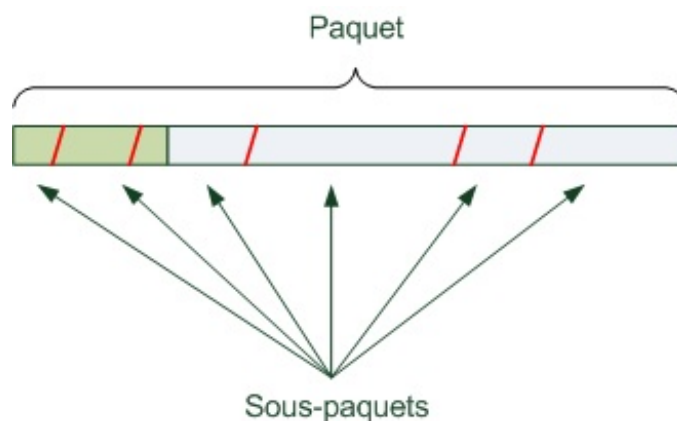
On demande à la socket combien d'octets ont été reçus dans le sous-paquet grâce à la méthode `bytesAvailable()`. Si on a reçu moins d'octets que la taille d'un `quint16`, on arrête la méthode de suite. On attendra le prochain appel de la méthode pour vérifier à nouveau si on a reçu assez d'octets pour récupérer la taille du message.

Code : C++

```
if (tailleMessage == 0) // Si on ne connaît pas encore la taille du  
message, on essaie de la récupérer  
{  
    if (socket->bytesAvailable() < (int)sizeof(quint16)) // On n'a  
pas reçu la taille du message en entier  
        return;  
  
    in >> tailleMessage; // Si on a reçu la taille du message en  
entier, on la récupère  
}
```

La ligne 6 est exécutée uniquement si on a reçu assez d'octets. En effet, le `return` a arrêté la méthode avant si ce n'était pas le cas. On récupère donc la taille du message et on la stocke. On la "retient" pour la suite des opérations.

Pour bien comprendre ce code, il faut se rappeler que le paquet est découpé en sous-paquets :



Notre slot est appelé à chaque fois qu'un sous-paquet a été reçu.

On vérifie si on a reçu assez d'octets pour récupérer la taille du message (première section en gris foncé). La taille de la première section "tailleMessage" peut être facilement retrouvée grâce à l'opérateur sizeof() que vous avez probablement déjà utilisé. Si on n'a pas reçu assez d'octets, on arrête la méthode (return). On attendra que le slot soit à nouveau appelé et on vérifiera alors cette fois si on a reçu assez d'octets.

Maintenant la suite des opérations. On a reçu la taille du message. On va maintenant essayer de récupérer le message lui-même :

Code : C++

```
// Si on connaît la taille du message, on vérifie si on a reçu le
// message en entier
if (socket->bytesAvailable() < tailleMessage) // Si on n'a pas
// encore tout reçu, on arrête la méthode
    return;
```

Le principe est le même. On regarde le nombre d'octets reçus, et si on en a moins que la taille annoncée du message, on arrête (return).

Si tout va bien, on peut passer à la suite de la méthode. Si ces lignes s'exécutent, c'est qu'on a reçu le message en entier, donc qu'on peut le récupérer dans une QString :

Code : C++

```
// Si ces lignes s'exécutent, c'est qu'on a reçu tout le message :
// on peut le récupérer !
QString message;
in >> message;
```

Notre QString "message" contient maintenant le message envoyé par le client !

Ouf ! Le serveur a reçu le message du client !

Mais ce n'est pas fini : il faut maintenant renvoyer le message à tous les clients comme je vous l'avais expliqué. Pour reprendre notre exemple, Vincent vient d'envoyer un message au serveur, celui-ci l'a récupéré et s'apprête à le renvoyer à tout le monde.

L'envoi du message à tout le monde se fait via la méthode envoyerATous dont je vous ai déjà parlé et qu'il va falloir écrire.

Code : C++

```
// 2 : on renvoie le message à tous les clients
```

```
envoyerATous (message) ;
```

On a presque fini. Il manque juste une petite chose : remettre `tailleMessage` à 0 pour que l'on puisse recevoir de futurs messages d'autres clients :

Code : C++

```
// 3 : remise de la taille du message à 0 pour permettre la
réception des futurs messages
tailleMessage = 0;
```

Si on n'avait pas fait ça, le serveur aurait cru lors du prochain sous-paquet reçu que le nouveau message est de la même longueur que le précédent, ce qui n'est certainement pas le cas. 😊

Bon, résumons le slot en entier :

Code : C++

```
void FenServeur::donneesRecues ()
{
    // 1 : on reçoit un paquet (ou un sous-paquet) d'un des clients

    // On détermine quel client envoie le message (recherche du
    QTcpSocket du client)
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à
    l'origine du signal, on arrête la méthode
        return;

    // Si tout va bien, on continue : on récupère le message
    QDataStream in(socket);

    if (tailleMessage == 0) // Si on ne connaît pas encore la
    taille du message, on essaie de la récupérer
    {
        if (socket->bytesAvailable() < (int)sizeof( quint16)) // On
        n'a pas reçu la taille du message en entier
            return;

        in >> tailleMessage; // Si on a reçu la taille du message
        en entier, on la récupère
    }

    // Si on connaît la taille du message, on vérifie si on a reçu
    le message en entier
    if (socket->bytesAvailable() < tailleMessage) // Si on n'a pas
    encore tout reçu, on arrête la méthode
        return;

    // Si ces lignes s'exécutent, c'est qu'on a reçu tout le
    message : on peut le récupérer !
    QString message;
    in >> message;

    // 2 : on renvoie le message à tous les clients
    envoyerATous (message) ;

    // 3 : remise de la taille du message à 0 pour permettre la
    réception des futurs messages
```

```
    tailleMessage = 0;
}
```

J'espère avoir été clair, car ce slot n'est pas simple et pas très facile à lire je dois bien avouer. La clé, le truc à comprendre, c'est que chaque return arrête la méthode. Le slot sera à nouveau appelé au prochain sous-paquet reçu, donc ces instructions s'exécuteront probablement plusieurs fois pour un message.

Si la méthode arrive à s'exécuter jusqu'au bout, c'est qu'on a reçu le message en entier. 😊

Slot deconnexionClient()

Ce slot est appelé lorsqu'un client se déconnecte.

On va envoyer un message à tous les clients encore connectés pour qu'ils sachent qu'un client vient de partir. Puis, on supprime le QTcpSocket correspondant au client dans notre tableau QList. Ainsi, le serveur "oublie" ce client, il ne considère plus qu'il fait partie des connectés.

Voici le slot en entier :

Code : C++

```
void FenServeur::deconnexionClient()
{
    envoyerATous(tr("<em>Un client vient de se déconnecter</em>"));

    // On détermine quel client se déconnecte
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à
l'origine du signal, on arrête la méthode
        return;

    clients.removeOne(socket);

    socket->deleteLater();
}
```

Comme plusieurs signaux sont connectés à ce slot, on ne sait pas quel est le client à l'origine de la déconnexion. Pour le retrouver, on utilise la même technique que pour le slot donneesRecues(), je ne la réexplique donc pas.

La méthode removeOne() de QList permet de supprimer le pointeur vers l'objet dans le tableau. Notre liste des clients est maintenant à jour.

Il ne reste plus qu'à finir de supprimer l'objet lui-même (nous venons seulement de supprimer le pointeur de la QList là). Pour supprimer l'objet, il faudrait faire un **delete** client;. Petit problème : si on supprime l'objet à l'origine du signal, on risque de faire bugger Qt. Heureusement tout a été prévu : on a juste à appeler deleteLater() (qui signifie "supprimer plus tard") et Qt se chargera de faire le delete lui-même un peu plus tard, lorsque notre slot aura fini de s'exécuter.

Méthode envoyerATous()

Ah, cette fois ce n'est pas un slot. 😊

C'est juste une méthode que j'ai décidé d'écrire dans la classe pour bien séparer le code, et aussi parce qu'on en a besoin plusieurs fois (vous avez remarqué que j'ai appelé cette méthode plusieurs fois dans les codes précédents non ?).

Dans le slot donneesRecues, nous recevons un message. Là, nous voulons au contraire en envoyer un, et ce à tous les clients connectés (tous les clients présents dans la QList).

Code : C++

```

void FenServeur::envoyerATous(const QString &message)
{
    // Préparation du paquet
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    out << (quint16) 0; // On écrit 0 au début du paquet pour
    réserver la place pour écrire la taille
    out << message; // On ajoute le message à la suite
    out.device()->seek(0); // On se replace au début du paquet
    out << (quint16) (paquet.size() - sizeof(quint16)); // On écrase
    le 0 qu'on avait réservé par la longueur du message

    // Envoi du paquet préparé à tous les clients connectés au
    serveur
    for (int i = 0; i < clients.size(); i++)
    {
        clients[i]->write(paquet);
    }
}

```

Quelques explications bien sûr. 😊

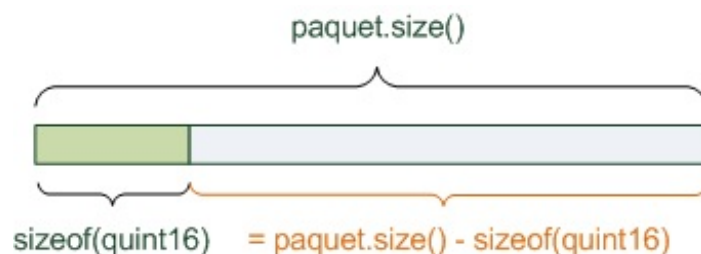
On crée un QByteArray "paquet" qui va contenir le paquet à envoyer sur le réseau. La classe QByteArray représente une suite d'octets quelconque.

On utilise un QDataStream comme tout à l'heure pour écrire dans le QByteArray facilement. Cela va nous permettre d'utiliser l'opérateur "<<".

Ce qui est particulier, c'est qu'on écrit d'abord le message (QString) et ensuite on calcule sa taille qu'on écrit au début du message.

Voilà ce qu'on fait sur le paquet dans l'ordre :

1. On écrit le nombre 0 de type quint16 pour "réserver" de la place.
2. On écrit à la suite le message, de type QString. Le message a été reçu en paramètre de la méthode envoyerATous().
3. On se replace au début du paquet (comme si on remettait le curseur au début d'un texte dans un traitement de texte).
4. On écrase le 0 qu'on avait écrit pour réserver de la place par la bonne taille du message. Cette taille est calculée via une simple soustraction : la taille du message est égale à la taille du paquet moins la taille réservée pour le quint16.



Notre paquet est prêt. Nous allons l'envoyer à tous les clients grâce à la méthode write() du socket. Pour cela, on fait une boucle sur la QList, et on envoie le message à chaque client.

Et voilà, le message est parti ! 😊

FenServeur.cpp en entier

Voici le contenu du fichier FenServeur.cpp que je viens de décortiquer en entier :

Code : C++

```
#include "FenServeur.h"

FenServeur::FenServeur()
{
    // Création et disposition des widgets de la fenêtre
    etatServeur = new QLabel;
    boutonQuitter = new QPushButton(tr("Quitter"));
    connect(boutonQuitter, SIGNAL(clicked()), qApp, SLOT(quit()));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(etatServeur);
    layout->addWidget(boutonQuitter);
    setLayout(layout);

    setWindowTitle(tr("ZeroChat - Serveur"));

    // Gestion du serveur
    serveur = new QTcpServer(this);
    if (!serveur->listen(QHostAddress::Any, 50885)) // Démarrage du
    serveur sur toutes les IP disponibles et sur le port 50585
    {
        // Si le serveur n'a pas été démarré correctement
        etatServeur->setText(tr("Le serveur n'a pas pu être démarré.
Raison :<br />") + serveur->errorString());
    }
    else
    {
        // Si le serveur a été démarré correctement
        etatServeur->setText(tr("Le serveur a été démarré sur le
port <strong>") + QString::number(serveur->serverPort()) +
tr("</strong>.<br />Des clients peuvent maintenant se connecter."));
        connect(serveur, SIGNAL(newConnection()), this,
SLOT(nouvelleConnexion()));
    }

    tailleMessage = 0;
}

void FenServeur::nouvelleConnexion()
{
    envoyerATous(tr("<em>Un nouveau client vient de se
connecter</em>"));

    QTcpSocket *nouveauClient = serveur->nextPendingConnection();
    clients << nouveauClient;

    connect(nouveauClient, SIGNAL(readyRead()), this,
SLOT(donneesRecues()));
    connect(nouveauClient, SIGNAL(disconnected()), this,
SLOT(deconnexionClient()));
}

void FenServeur::donneesRecues()
{
    // 1 : on reçoit un paquet (ou un sous-paquet) d'un des clients

    // On détermine quel client envoie le message (recherche du
QTcpSocket du client)
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à
l'origine du signal, on arrête la méthode
        return;

    // Si tout va bien, on continue : on récupère le message
```

```

        QDataStream in(socket);

        if (tailleMessage == 0) // Si on ne connaît pas encore la
taille du message, on essaie de la récupérer
        {
            if (socket->bytesAvailable() < (int)sizeof(quint16)) // On
n'a pas reçu la taille du message en entier
                return;

            in >> tailleMessage; // Si on a reçu la taille du message
en entier, on la récupère
        }

        // Si on connaît la taille du message, on vérifie si on a reçu
le message en entier
        if (socket->bytesAvailable() < tailleMessage) // Si on n'a pas
encore tout reçu, on arrête la méthode
            return;

        // Si ces lignes s'exécutent, c'est qu'on a reçu tout le
message : on peut le récupérer !
        QString message;
        in >> message;

        // 2 : on renvoie le message à tous les clients
        envoyerATous(message);

        // 3 : remise de la taille du message à 0 pour permettre la
réception des futurs messages
        tailleMessage = 0;
    }

void FenServeur::deconnexionClient()
{
    envoyerATous(tr("<em>Un client vient de se déconnecter</em>"));

    // On détermine quel client se déconnecte
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à
l'origine du signal, on arrête la méthode
        return;

    clients.removeOne(socket);

    socket->deleteLater();
}

void FenServeur::envoyerATous(const QString &message)
{
    // Préparation du paquet
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    out << (quint16) 0; // On écrit 0 au début du paquet pour
réserver la place pour écrire la taille
    out << message; // On ajoute le message à la suite
    out.device()->seek(0); // On se replace au début du paquet
    out << (quint16) (paquet.size() - sizeof(quint16)); // On écrase
le 0 qu'on avait réservé par la longueur du message

    // Envoi du paquet préparé à tous les clients connectés au
serveur
    for (int i = 0; i < clients.size(); i++)
    {
        clients[i]->write(paquet);
    }
}

```

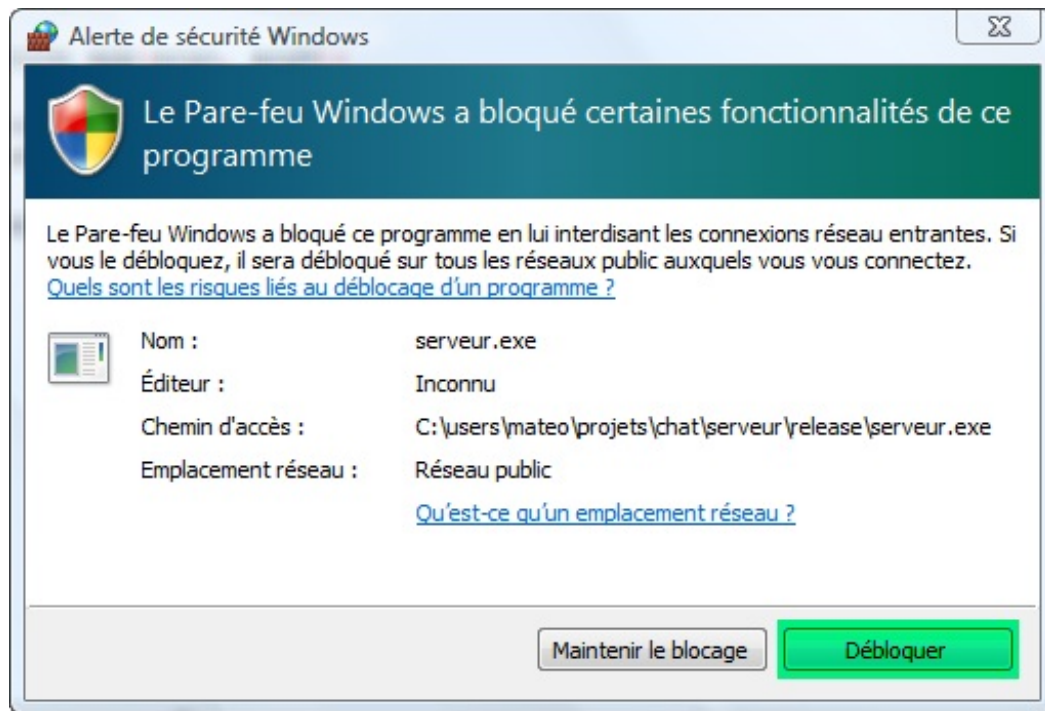
```
}
```

Lancement du serveur

Bonne nouvelle, devinez quoi : notre projet "serveur" est terminé !

Nous avons fait le plus dur, l'implémentation du serveur dans FenServeur.cpp. Compilez, et lancez le serveur ainsi créé.

Vous risquez d'avoir une alerte de votre pare-feu (firewall). Par exemple sous Windows :



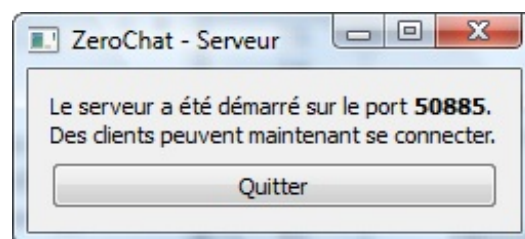
En effet, notre programme va communiquer sur le réseau. Le pare-feu nous demande si nous voulons autoriser notre programme à le faire : répondez oui en cliquant sur "Débloquer".



Dans cet exemple, j'ai considéré que le pare-feu de votre système d'exploitation était votre seul pare-feu. Si vous comptez utiliser le Chat sur internet et que vous êtes derrière un routeur, vérifiez la configuration du routeur et ouvrez le port 50885 pour les données TCP.

Si le Chat n'a pas l'air de fonctionner, c'est très probablement à cause d'un pare-feu quelque part qui bloque le port.

Notre serveur est maintenant lancé :



Bravo ! 😊

Laissez ce programme tourner en fond sur votre ordinateur (vous pouvez réduire la fenêtre). Il va servir à faire la communication entre les différents clients.

Bon, mauvaise nouvelle chers auditeurs : nous avons beaucoup sué, mais nous avons fait seulement 50% du travail ! Il faut maintenant s'attaquer au projet "client" pour réaliser le programme qui sera utilisé par tous les clients pour chatter.

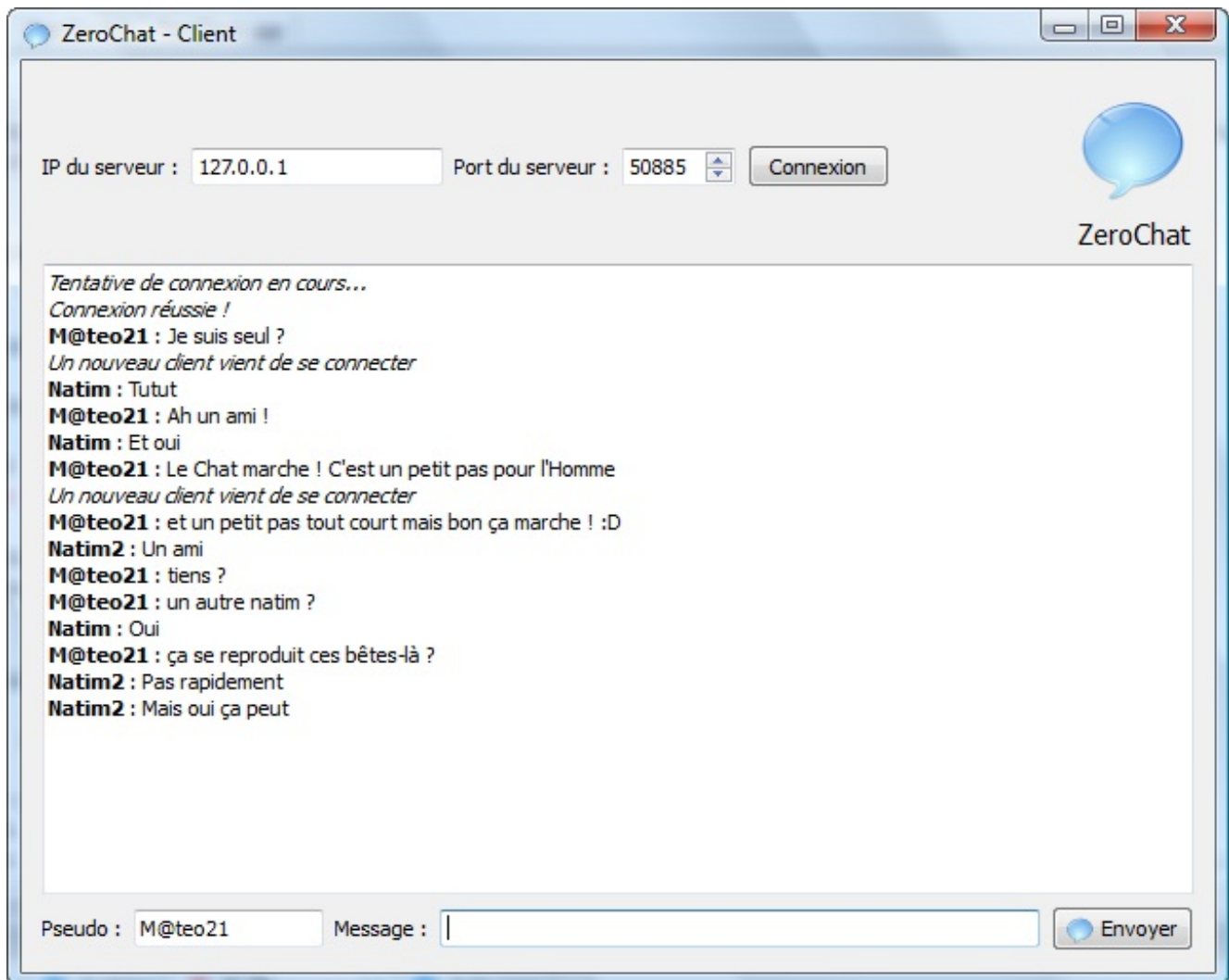
Heureusement, nous avons déjà fait le plus dur en analysant le slot `donneesRecues`, on devrait donc aller un peu plus vite. 😊

Réalisation du client

Si la fenêtre du serveur était toute simple, il en va autrement pour la fenêtre du client.

En effet, autant créer une fenêtre pour le serveur était facultatif, autant pour le client il faut bien qu'il ait une fenêtre pour écrire ses messages. 😊

Voici la fenêtre de client que l'on veut coder :



Nous aurons 3 fichiers à nouveau :

- main.cpp
- FenClient.h
- FenClient.cpp

Dessin de la fenêtre avec Qt Designer

Bon, la réalisation de cette fenêtre ne nous intéresse pas vraiment. C'est une fenêtre tout ce qu'il y a de plus classique.

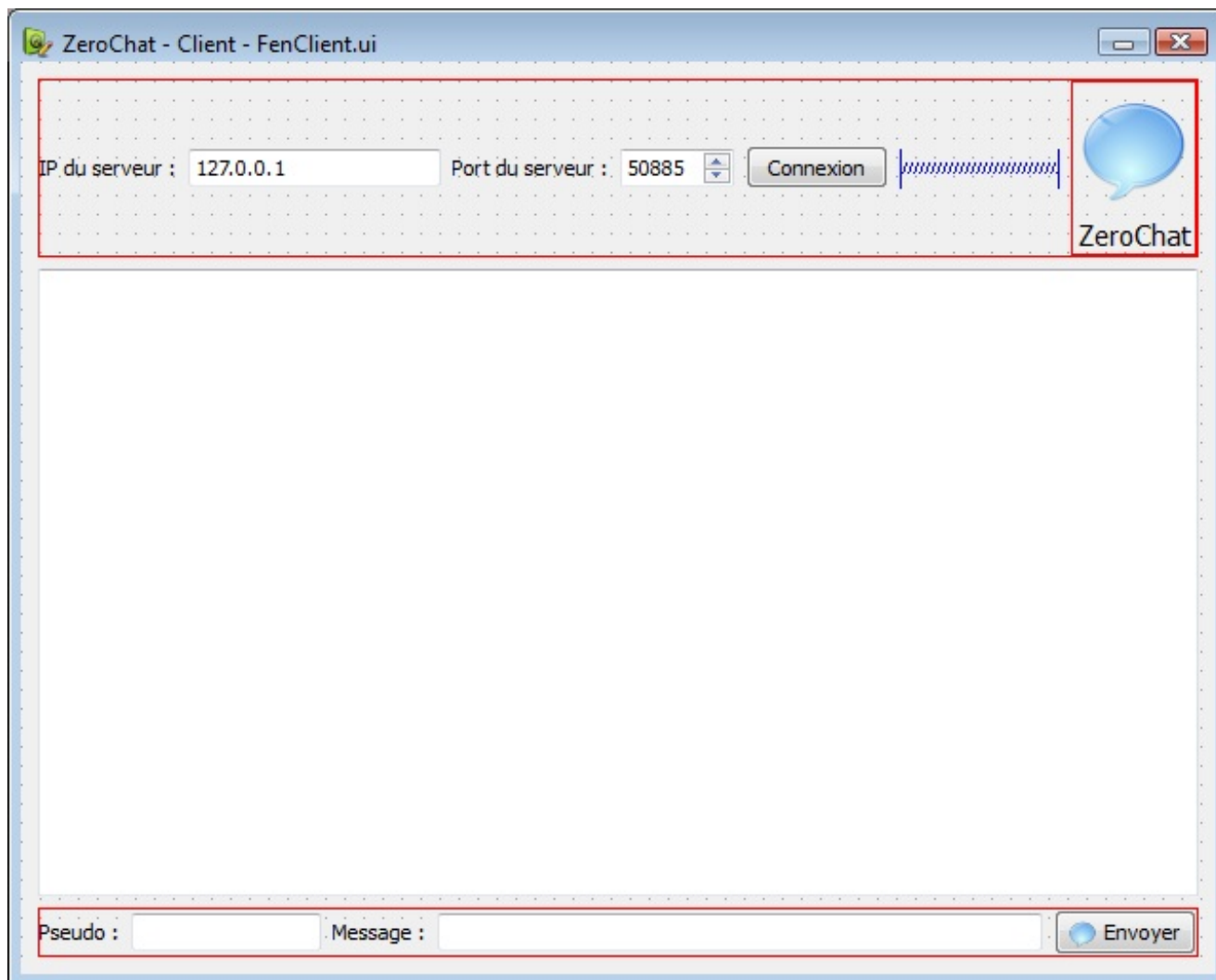
Pour gagner du temps, je vous propose de créer l'interface de la fenêtre du client via Qt Designer. Ce programme a justement été conçu pour gagner du temps pour ceux qui savent déjà coder la fenêtre à la main, ce qui est notre cas.



J'aurais pu utiliser Qt Designer pour le TP zNavigo aussi, mais la fenêtre était beaucoup plus complexe et le nombre d'onglets variable. J'avais donc décidé de l'écrire à la main.

Pour une fenêtre assez simple comme celle-là, l'utilisation de Qt Designer permet au contraire de gagner du temps, pour peu qu'on sache déjà coder la fenêtre à la main.

J'ouvre donc Qt Designer et je dessine la fenêtre suivante :



Je prends soin à bien donner un nom correct à chacun des widgets via la propriété `objectName` (sauf pour les `QLabel`, car on n'aura pas à les réutiliser ceux-là donc on s'en moque un peu 😊).

Je veille aussi à utiliser des layouts partout sur ma fenêtre pour la rendre redimensionnable sans problème. Je sélectionne plusieurs widgets à la fois et je clique sur un des boutons de la barre d'outils en haut pour les assembler selon un layout (notez que je n'utilise jamais les layouts de la widget box à gauche).

Pour vous faire gagner du temps si vous ne voulez pas redessiner la fenêtre sous Qt Designer, voici le fichier `FenClient.ui` que j'ai généré :

[Télécharger FenClient.ui](#)
(faites clic droit / enregistrer sous)

Client.pro

J'ai mis à jour le fichier `.pro` pour indiquer qu'il y avait un UI dans le projet et qu'on utilisait le module `network` de Qt. Vérifiez donc que votre `.pro` ressemble au mien :

Code : Autre

```
#####
# Automatically generated by qmake (2.01a) mer. 25. juin 17:13:47 2008
```

```
#####

TEMPLATE = app
QT += network
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenClient.h
FORMS += FenClient.ui
SOURCES += FenClient.cpp main.cpp
```

main.cpp

Revenons au code. Le main est toujours très simple et sans originalité : il ouvre la fenêtre.

Code : C++

```
#include <QApplication>
#include "FenClient.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenClient fenetre;
    fenetre.show();

    return app.exec();
}
```

FenClient.h

Notre fenêtre utilise un fichier généré avec Qt Designer. Direction le chapitre sur Qt Designer si vous avez oublié comment se servir d'une fenêtre générée dans son code.

Je vais ici fonctionner un peu différemment en utilisant un héritage multiple, une notion que nous n'avons pas vue précédemment mais qui est assez simple à comprendre : la classe va hériter de 2 classes. L'intérêt de ce double héritage est d'éviter de devoir mettre le préfixe `ui->` partout dans le code. Bien sûr, vous pouvez faire comme vous avez appris dans le chapitre Qt Designer si cela vous perturbe.

Code : C++

```
#ifndef HEADER_FENCLIENT
#define HEADER_FENCLIENT

#include <QtGui>
#include <QtNetwork>
#include "ui_FenClient.h"

class FenClient : public QWidget, private Ui::FenClient
{
    Q_OBJECT

public:
    FenClient();
}
```

```

private slots:
    void on_boutonConnexion_clicked();
    void on_boutonEnvoyer_clicked();
    void on_message_returnPressed();
    void donneesRecues();
    void connecte();
    void deconnecte();
    void erreurSocket(QAbstractSocket::SocketError erreur);

private:
    QTcpSocket *socket; // Représente le serveur
    quint16 tailleMessage;
};

#endif

```

Notez qu'on inclut `ui_FenClient.h` pour réutiliser la fenêtre générée.

Notre fenêtre comporte pas mal de slots qu'il va falloir implémenter, heureusement ils seront assez simples. On utilise les `autoconnect` pour les 3 premiers d'entre eux pour gérer les événements de la fenêtre :

- **`on_boutonConnexion_clicked()`** : appelé lorsqu'on clique sur le bouton "Connexion" et qu'on souhaite donc se connecter au serveur.
- **`on_boutonEnvoyer_clicked()`** : appelé lorsqu'on clique sur "Envoyer" pour envoyer un message dans le Chat.
- **`on_message_returnPressed()`** : appelé lorsqu'on appuie sur la touche "Entrée" lorsqu'on rédige un message. Comme cela revient au même que `on_boutonEnvoyer_clicked()`, on appellera cette méthode directement pour éviter d'avoir à écrire 2 fois le même code.
- **`donneesRecues()`** : appelé lorsqu'on reçoit un sous-paquet du serveur. Ce slot sera très similaire à celui du serveur qui possède le même nom.
- **`connecte()`** : appelé lorsqu'on vient de réussir à se connecter au serveur.
- **`deconnecte()`** : appelé lorsqu'on vient de se déconnecter du serveur.
- **`erreurSocket(QAbstractSocket::SocketError erreur)`** : appelé lorsqu'il y a eu une erreur sur le réseau (connexion au serveur impossible par exemple).

En plus de ça, on a 2 attributs à manipuler :

- **`QTcpSocket *socket`** : une socket qui représentera la connexion au serveur. On utilisera cette socket pour envoyer des paquets au serveur, par exemple lorsque l'utilisateur veut envoyer un message.
- **`quint16 tailleMessage`** : permet à l'objet de se "souvenir" de la taille du message qu'il est en train de recevoir dans son slot `donneesRecues()`. Il a la même utilité que sur le serveur.

FenClient.cpp

Maintenant implémentons tout ce beau monde !

Courage, après ça c'est fini vous allez pouvoir savourer votre Chat ! 😊

Le constructeur

Notre constructeur se doit d'appeler `setupUi()` dès le début pour mettre en place les widgets sur la fenêtre. C'est justement là qu'on gagne du temps grâce à Qt Designer : on n'a pas à coder le placement des widgets sur la fenêtre. 😊

Code : C++

```
FenClient::FenClient()
```

```

{
    setupUi (this);

    socket = new QTcpSocket (this);
    connect (socket, SIGNAL (readyRead()), this,
    SLOT (donneesRecues()));
    connect (socket, SIGNAL (connected()), this, SLOT (connecte()));
    connect (socket, SIGNAL (disconnected()), this,
    SLOT (deconnecte()));
    connect (socket, SIGNAL (error (QAbstractSocket::SocketError)),
    this, SLOT (erreurSocket (QAbstractSocket::SocketError)));

    tailleMessage = 0;
}

```

En plus de `setupUi()`, on fait quelques initialisations supplémentaires indispensables :

- On crée l'objet de type `QTcpSocket` qui va représenter la connexion au serveur.
- On connecte les signaux qu'il est susceptible d'envoyer à nos slots personnalisés.
- On met `tailleMessage` à 0 pour permettre la réception de nouveaux messages.

Notez qu'on ne se connecte pas au serveur dans le constructeur. On prépare juste la socket, mais on ne fera la connexion que lorsque le client aura cliqué sur le bouton "Connexion" (il faut bien lui laisser le temps de rentrer l'adresse IP du serveur !).

Slot on_boutonConnexion_clicked()

Ce slot se fait appeler dès que l'on a cliqué sur le bouton "Connexion" en haut de la fenêtre.

Code : C++

```

// Tentative de connexion au serveur
void FenClient::on_boutonConnexion_clicked()
{
    // On annonce sur la fenêtre qu'on est en train de se connecter
    listeMessages->append(tr("<em>Tentative de connexion en
cours...</em>"));
    boutonConnexion->setEnabled(false);

    socket->abort(); // On désactive les connexions précédentes s'il
y en a
    socket->connectToHost(serveurIP->text(), serveurPort->value());
    // On se connecte au serveur demandé
}

```

1. Dans un premier temps, on affiche sur la zone de messages "listeMessages" au centre de la fenêtre que l'on est en train d'essayer de se connecter.
2. On désactive temporairement le bouton "Connexion" pour empêcher au client de retenter une connexion alors qu'une tentative de connexion est déjà en cours.
3. Si la socket est déjà connectée à un serveur, on coupe la connexion avec `abort()`. Si on n'était pas connecté, cela n'aura aucun effet, mais c'est par sécurité pour que l'on ne soit pas connecté à 2 serveurs à la fois.
4. On se connecte enfin au serveur avec la méthode `connectToHost()`. On utilise l'IP et le port demandés par l'utilisateur dans les champs de texte en haut de la fenêtre.

Slot on_boutonEnvoyer_clicked()

Ce slot est appelé lorsqu'on essaie d'envoyer un message (le bouton "Envoyer" en bas à droite a été cliqué).

Code : C++

```
// Envoi d'un message au serveur
void FenClient::on_boutonEnvoyer_clicked()
{
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    // On prépare le paquet à envoyer
    QString messageAEnvoyer = tr("<strong>") + pseudo->text()
+tr("</strong> : ") + message->text();

    out << (quint16) 0;
    out << messageAEnvoyer;
    out.device()->seek(0);
    out << (quint16) (paquet.size() - sizeof(quint16));

    socket->write(paquet); // On envoie le paquet

    message->clear(); // On vide la zone d'écriture du message
    message->setFocus(); // Et on remet le curseur à l'intérieur
}
```

Ce code est similaire à celui de la méthode envoyerATous() du serveur. Il s'agit d'un envoi de données sur le réseau.

1. On prépare un QByteArray dans lequel on va écrire le paquet qu'on veut envoyer.
2. On construit ensuite la QString contenant le message à envoyer. Vous noterez qu'on met le nom de l'auteur et son texte directement dans la même QString. Idéalement, il vaudrait mieux séparer les deux pour avoir un code plus logique et plus modulable, mais cela aurait compliqué le code de ce chapitre bien délicat, donc ça sera dans les améliorations à faire à la fin. 🤔
3. On calcule la taille du message.
4. On envoie le paquet ainsi créé au serveur en utilisant la socket qui le représente et sa méthode write().
5. On efface automatiquement la zone d'écriture des messages en bas pour qu'on puisse en écrire un nouveau et on donne le focus à cette zone immédiatement pour que le curseur soit placé dans le bon widget.

Slot on_message_returnPressed()

Ce slot est appelé lorsqu'on a appuyé sur la touche "Entrée" après avoir rédigé un message.

Cela a le même effet qu'un clic sur le bouton "Envoyer", nous appelons donc le slot que nous venons d'écrire :

Code : C++

```
// Appuyer sur la touche Entrée a le même effet que cliquer sur le
bouton "Envoyer"
void FenClient::on_message_returnPressed()
{
    on_boutonEnvoyer_clicked();
}
```

Slot donneesRecues()

Voilà à nouveau le slot de vos pires cauchemars. 🐼

Il est quasiment identique à celui du serveur (la réception de données fonctionne de la même manière) je ne le réexplique donc pas :

Code : C++

```
// On a reçu un paquet (ou un sous-paquet)
void FenClient::donneesRecues()
{
    /* Même principe que lorsque le serveur reçoit un paquet :
    On essaie de récupérer la taille du message
    Une fois qu'on l'a, on attend d'avoir reçu le message entier (en se
    basant sur la taille annoncée tailleMessage)
    */
    QDataStream in(socket);

    if (tailleMessage == 0)
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16))
            return;

        in >> tailleMessage;
    }

    if (socket->bytesAvailable() < tailleMessage)
        return;

    // Si on arrive jusqu'à cette ligne, on peut récupérer le
    message entier
    QString messageRecu;
    in >> messageRecu;

    // On affiche le message sur la zone de Chat
    listeMessages->append(messageRecu);

    // On remet la taille du message à 0 pour pouvoir recevoir de
    futurs messages
    tailleMessage = 0;
}
```

La seule différence ici en fait, c'est qu'on affiche le message reçu dans la zone de Chat à la fin : `listeMessages->append(messageRecu);`

Slot connecte()

Ce slot est appelé lorsqu'on a réussi à se connecter au serveur.

Code : C++

```
// Ce slot est appelé lorsque la connexion au serveur a réussi
void FenClient::connecte()
{
    listeMessages->append(tr("<em>Connexion réussie !</em>"));
    boutonConnexion->setEnabled(true);
}
```

Tout bêtement, on se contente d'afficher "Connexion réussie" dans la zone de Chat pour que le client sache qu'il est bien connecté au serveur.

On réactive aussi le bouton "Connexion" qu'on avait désactivé, pour permettre une nouvelle connexion à un autre serveur.

Slot deconnecte()

Ce slot est appelé lorsqu'on est déconnecté du serveur.

Code : C++

```
// Ce slot est appelé lorsqu'on est déconnecté du serveur
void FenClient::deconnecte()
{
    listeMessages->append(tr("<em>Déconnecté du serveur</em>"));
}
```

On affiche juste un message sur la zone de texte pour que le client soit au courant.

Slot erreurSocket()

Ce slot est appelé lorsque la socket a rencontré une erreur.

Code : C++

```
// Ce slot est appelé lorsqu'il y a une erreur
void FenClient::erreurSocket(QAbstractSocket::SocketError erreur)
{
    switch(erreur) // On affiche un message différent selon
l'erreur qu'on nous indique
    {
        case QAbstractSocket::HostNotFoundError:
            listeMessages->append(tr("<em>ERREUR : le serveur n'a
pas pu être trouvé. Vérifiez l'IP et le port.</em>"));
            break;
        case QAbstractSocket::ConnectionRefusedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a
refusé la connexion. Vérifiez si le programme \"serveur\" a bien été
lancé. Vérifiez aussi l'IP et le port.</em>"));
            break;
        case QAbstractSocket::RemoteHostClosedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a
coupé la connexion.</em>"));
            break;
        default:
            listeMessages->append(tr("<em>ERREUR : ") + socket-
>errorString() + tr("</em>"));
    }

    boutonConnexion->setEnabled(true);
}
```

La raison de l'erreur est passée en paramètre. Elle est de type `QAbstractSocket::SocketError` (c'est une énumération).

On fait un switch pour afficher un message différent en fonction de l'erreur. Je n'ai pas traité toutes les erreurs possibles, lisez la doc pour connaître les autres raisons d'erreurs que l'on peut gérer.

La plupart des erreurs que je gère ici sont liées à la connexion au serveur. J'affiche un message intelligible en français pour que l'on comprenne la raison de l'erreur.

Le cas "default" est appelé pour les erreurs que je n'ai pas gérées. J'affiche le message d'erreur envoyé par la socket (qui sera peut-être en anglais mais bon c'est mieux que rien).

FenClient.cpp en entier

C'est fini ! 😊

Bon, ça n'a pas été trop long ni trop difficile après avoir fait le serveur, avouez 😊

Voici le code complet de FenClient.cpp :

Code : C++

```
#include "FenClient.h"

FenClient::FenClient()
{
    setupUi(this);

    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(readyRead()), this,
    SLOT(donneesRecues()));
    connect(socket, SIGNAL(connected()), this, SLOT(connecte()));
    connect(socket, SIGNAL(disconnected()), this,
    SLOT(deconnecte()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)),
    this, SLOT(erreurSocket(QAbstractSocket::SocketError)));

    tailleMessage = 0;
}

// Tentative de connexion au serveur
void FenClient::on_boutonConnexion_clicked()
{
    // On annonce sur la fenêtre qu'on est en train de se connecter
    listeMessages->append(tr("<em>Tentative de connexion en
cours...</em>"));
    boutonConnexion->setEnabled(false);

    socket->abort(); // On désactive les connexions précédentes s'il
y en a
    socket->connectToHost(serveurIP->text(), serveurPort->value());
    // On se connecte au serveur demandé
}

// Envoi d'un message au serveur
void FenClient::on_boutonEnvoyer_clicked()
{
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    // On prépare le paquet à envoyer
    QString messageAEnvoyer = tr("<strong>") + pseudo->text()
+tr("</strong> : ") + message->text();

    out << (quint16) 0;
    out << messageAEnvoyer;
    out.device()->seek(0);
    out << (quint16) (paquet.size() - sizeof(quint16));

    socket->write(paquet); // On envoie le paquet

    message->clear(); // On vide la zone d'écriture du message
    message->setFocus(); // Et on remet le curseur à l'intérieur
}

// Appuyer sur la touche Entrée a le même effet que cliquer sur le
bouton "Envoyer"
void FenClient::on_message_returnPressed()
{
    on_boutonEnvoyer_clicked();
}
```

```

// On a reçu un paquet (ou un sous-paquet)
void FenClient::donneesRecues()
{
    /* Même principe que lorsque le serveur reçoit un paquet :
    On essaie de récupérer la taille du message
    Une fois qu'on l'a, on attend d'avoir reçu le message entier (en se
    basant sur la taille annoncée tailleMessage)
    */
    QDataStream in(socket);

    if (tailleMessage == 0)
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16))
            return;

        in >> tailleMessage;
    }

    if (socket->bytesAvailable() < tailleMessage)
        return;

    // Si on arrive jusqu'à cette ligne, on peut récupérer le
    message entier
    QString messageRecu;
    in >> messageRecu;

    // On affiche le message sur la zone de Chat
    listeMessages->append(messageRecu);

    // On remet la taille du message à 0 pour pouvoir recevoir de
    futurs messages
    tailleMessage = 0;
}

// Ce slot est appelé lorsque la connexion au serveur a réussi
void FenClient::connecte()
{
    listeMessages->append(tr("<em>Connexion réussie !</em>"));
    boutonConnexion->setEnabled(true);
}

// Ce slot est appelé lorsqu'on est déconnecté du serveur
void FenClient::deconnecte()
{
    listeMessages->append(tr("<em>Déconnecté du serveur</em>"));
}

// Ce slot est appelé lorsqu'il y a une erreur
void FenClient::erreurSocket(QAbstractSocket::SocketError erreur)
{
    switch(erreur) // On affiche un message différent selon
    l'erreur qu'on nous indique
    {
        case QAbstractSocket::HostNotFoundError:
            listeMessages->append(tr("<em>ERREUR : le serveur n'a
pas pu être trouvé. Vérifiez l'IP et le port.</em>"));
            break;
        case QAbstractSocket::ConnectionRefusedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a
refusé la connexion. Vérifiez si le programme \"serveur\" a bien été
lancé. Vérifiez aussi l'IP et le port.</em>"));
            break;
        case QAbstractSocket::RemoteHostClosedError:
            listeMessages->append(tr("<em>ERREUR : le serveur a
coupé la connexion.</em>"));
            break;
        default:
            listeMessages->append(tr("<em>ERREUR : ") + socket-
>errorString() + tr("</em>"));
    }
}

```

```
    }  
  
    boutonConnexion->setEnabled(true);  
}
```

Test du Chat et améliorations

~~Notre projet~~ Nos projets sont terminés !
Nous avons fait le client et le serveur !

Je vous propose de tester le bon fonctionnement du Chat dans un premier temps, et éventuellement de télécharger les projets tous prêts.

Nous verrons ensuite comment vous pouvez améliorer tout cela. 😊

Tester le Chat

Avant toute chose, vous voudrez peut-être récupérer le projet tout prêt et zippé pour partir sur la même base que moi.

[Télécharger Chat.zip \(60 Ko\)](#)

Le zip contient un sous-dossier par projet : serveur et client.

Vous pouvez exécuter directement les programmes serveur.exe et client.exe si vous êtes sous Windows (en n'oubliant pas de mettre les DLL de Qt dans le même répertoire). Si vous utilisez un autre OS, vous devrez recompiler le projet (faites un qmake et un make).

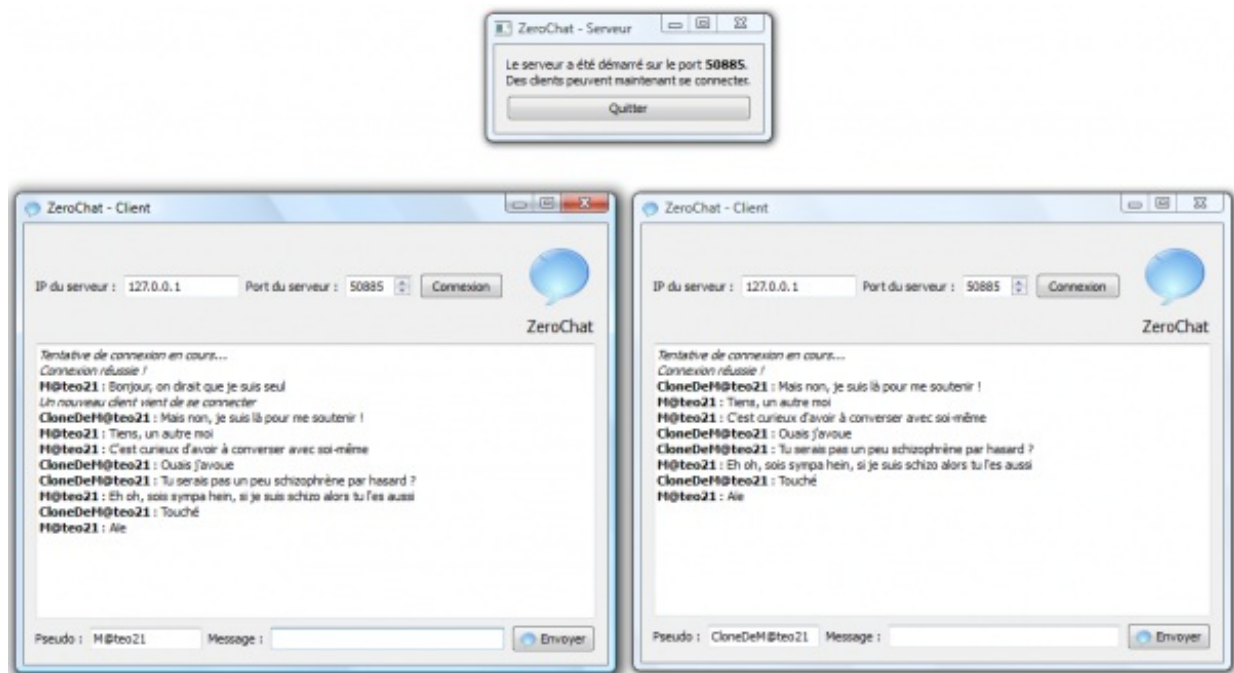
Vous pouvez dans un premier temps tester le Chat en interne sur votre propre ordinateur. Je vous propose de lancer :

- Un serveur
- Deux clients

Cela va nous permettre de simuler une conversation en interne sur notre ordinateur. Cela utilisera le réseau, mais à l'intérieur de votre propre machine. 😊

J'avoue que c'est un peu curieux, mais si ça fonctionne en interne, ça fonctionnera en réseau local et sur internet sans problème (pour peu que le port soit ouvert). C'est donc une bonne idée de faire ses tests en interne dans un premier temps.

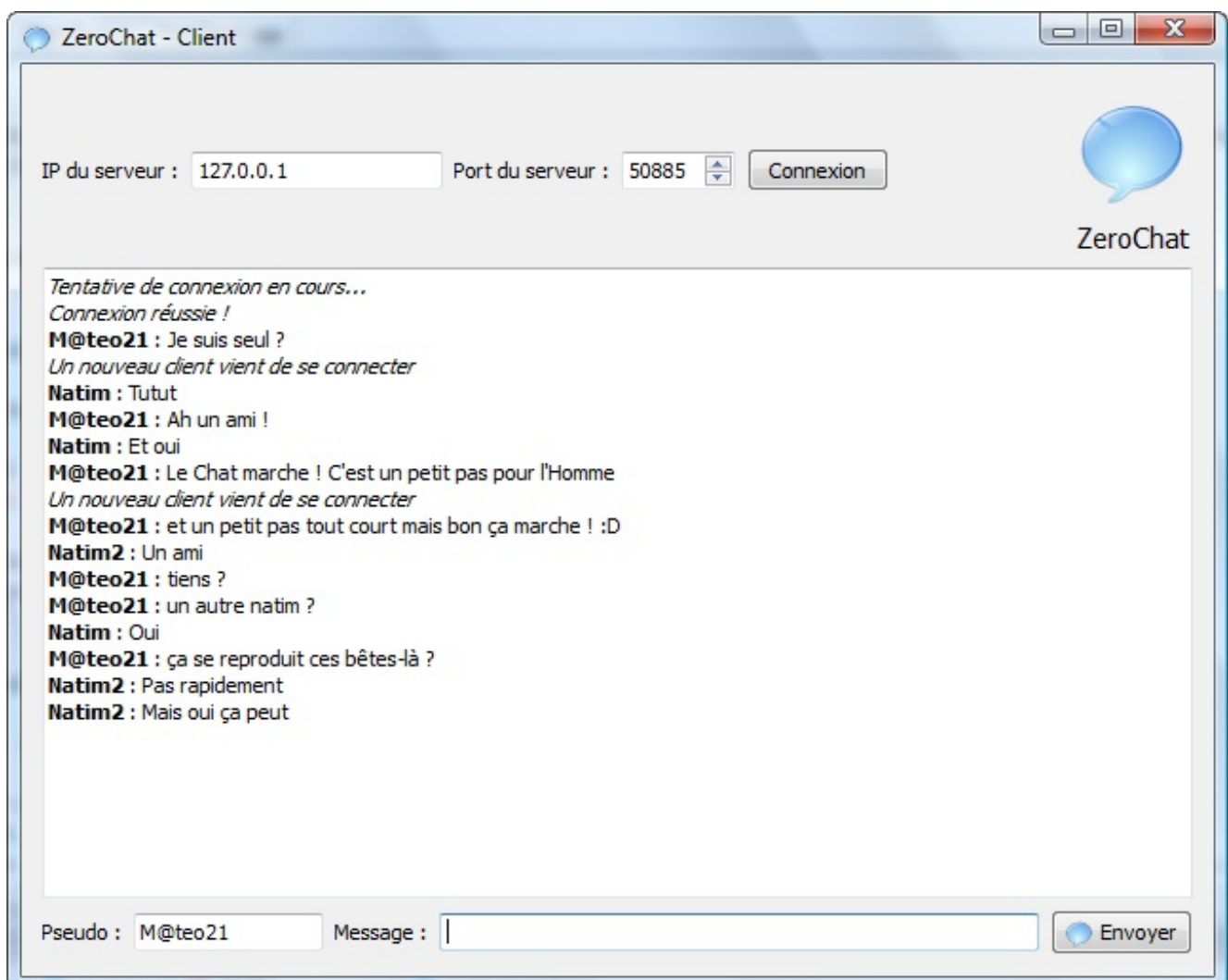
Voici ce que ça donne quand je me parle à moi-même :



Comme vous pouvez le voir, tout fonctionne (sauf peut-être mon cerveau 🤔).

Amusez-vous ensuite à tester le programme en réseau local ou sur internet avec des amis. Pensez à chaque fois à vérifier si le port est ouvert. Si vous avez un problème avec le programme, il y a 99% de chances que ça vienne du port.

Voici une petite conversation en réseau local :



Je ne l'ai pas testé sur internet mais je sais pertinemment que ça fonctionne. Le principe du réseau est le même partout, que ce soit en interne, en local ou via internet. C'est juste l'IP qui change à chaque fois.

Améliorations à réaliser

Bon, le moins qu'on puisse dire c'est que j'ai bien travaillé dans ce chapitre, maintenant à votre tour de bosser un peu. 😊

Voici quelques suggestions d'améliorations que vous pouvez réaliser sur le Chat, plus ou moins difficiles selon le cas :

- Vous pouvez **griser la zone d'envoi des messages** ainsi que le bouton "Envoyer" lorsque le client n'est pas connecté.
- Sur la fenêtre du serveur, il devrait être assez facile d'**afficher le nombre de clients qui sont connectés**.
- Plus délicat car ça demande un peu de réorganisation du code : au lieu d'avoir une QList de QTcpSocket, faites une QList d'objets de type Client.
Il faudra **créer une nouvelle classe Client** qui va représenter un client. Elle aura des attributs comme : sa QTcpSocket, le pseudo (QString), pourquoi pas l'avatar (QPixmap), etc. A partir de là vous aurez alors beaucoup plus de souplesse dans votre Chat !
- Vous pourriez alors facilement **afficher le pseudo du membre qui vient se connecter**. Pour le moment, on a juste "Un client vient de se connecter".
- Plutôt graphique mais sympa : vous pourriez **gérer la mise en forme des messages** (gras, rouge...) ainsi que des **smilies**. Bon après il s'agit pas de recréer MSN (quoique, le principe est tout à fait le même 😊) donc n'allez pas trop loin dans ce genre de fonctionnalités quand même.
- Plus délicat, mais très intéressant : essayez d'**afficher la liste des clients connectés** sur la fenêtre des clients. Vous devriez rajouter un widget QListView pour afficher cette liste, ça vous ferait travailler MVC en plus. 😊
Le plus délicat est de gérer la liste des connectés, car pour le moment le pseudo est directement intégré aux messages qui sont envoyés. Il faudrait essayer de gérer le contenu des paquets un peu différemment, à vous de voir.
- Actuellement, le serveur est un peu minimal et ne gère pas tous les cas. Par exemple, si 2 clients envoient un message en même temps, il n'y a qu'une seule variable tailleMessage pour 2 messages en cours de réception. Je vous recommande de gérer plutôt **1 tailleMessage par client** (vous n'avez qu'à mettre tailleMessage dans la classe Client).

Je m'arrête là pour les suggestions, il y a déjà du travail !

On pourrait aussi imaginer de permettre un Chat en privé entre certains clients, ou encore d'autoriser l'envoi de fichier sur le réseau (le tout étant de récupérer le fichier à envoyer sous forme de QByteArray).

Enfin, n'oubliez pas que le réseau ne se limite pas au Chat. Si vous faites un jeu en réseau par exemple, il faudra non pas envoyer des messages texte, mais plutôt les actions des autres joueurs. Dans ce cas, le schéma des paquets envoyés deviendra un peu plus complexe, mais c'est nécessaire.

A vous d'adapter un peu mon code, vous êtes grands maintenant, au boulot ! 😊

Bon, je crois que c'était mon plus gros chapitre. Il était temps que je m'arrête. 😊

J'espère que vous avez apprécié cette partie sur Qt, nous avons vu beaucoup de choses (un peu trop même) et pourtant nous n'avons pas tout vu. 😊

Je vous laisse vous entraîner avec le réseau, les widgets, et pour tout le reste n'oubliez pas : il y a la doc ! Et les forums du Site du Zéro aussi, oui oui, c'est vrai.

Partie 4 : [Théorie] Utilisez la bibliothèque standard

Dans les deux premières parties du cours, nous avons appris les bases du langage et dans la troisième, nous avons pu créer des fenêtres (et beaucoup d'autres choses) pour nos programmes. Il est donc temps maintenant de replonger un peu dans la théorie pour découvrir la bibliothèque standard du C++. Vous en connaissez déjà certaines parties telles que `cout`, `fstream` ou `vector`. Construire vos programmes deviendra plus simple grâce aux nombreux outils proposés.

Qu'est-ce que la bibliothèque standard ?

Maintenant que vous êtes des champions de Qt, découvrir une bibliothèque de fonctions ne devrait pas vous faire peur. Vous verrez qu'utiliser les outils standards n'est pas toujours de tout repos, mais cela peut rendre vos programmes diablement plus simples à écrire et plus efficaces.

La bibliothèque standard du C++ est la **bibliothèque officielle du langage**, c'est-à-dire qu'elle est disponible partout où l'on peut utiliser le C++. Apprendre à l'utiliser vous permettra de travailler même sur les plateformes les plus excentriques où d'autres outils, comme Qt par exemple, n'existent tout simplement pas.

Ce chapitre d'introduction à la bibliothèque standard (la SL) ne devrait pas vous poser de problèmes de compréhension. On va commencer en douceur par se cultiver un peu et revoir certains éléments dont vous avez déjà entendu parler.

Nous allons dans le chapitre suivant nous plonger réellement dans la partie intéressante de la bibliothèque, la célèbre STL.

Cette première introduction devrait vous mettre dans le bain et vous permettre de mieux appréhender la suite.

Un peu d'histoire

Dans l'introduction de ce cours, je vous ai déjà un petit peu parlé de l'histoire des langages de programmation en général afin de situer le C++. Je vous propose d'entrer un peu plus dans les détails pour comprendre pourquoi un langage possède une bibliothèque standard.

La petite histoire raccourcie du C++

Prenons notre machine à remonter le temps et retournons à l'époque de l'informatique où le CD n'existait pas, où la souris n'existait pas, où les ordinateurs étaient moins puissants que les processeurs de votre lave-linge ou de votre four micro-ondes...

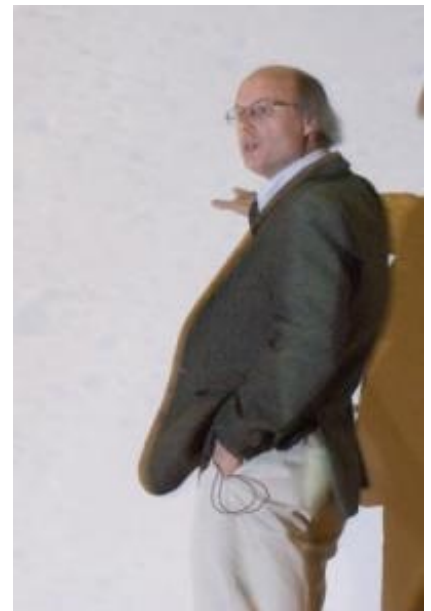
Nous sommes en 1979, Bjarne Stroustrup, un informaticien de chez AT&T, développe le "C with classes" à partir du C et en s'inspirant des langages plus modernes et innovants de l'époque comme le [Simula](#). Il écrit lui-même le premier compilateur de son nouveau langage et l'utilise dans son travail. A l'époque, son langage n'était utilisé que par lui-même pour ses recherches personnelles. Il voulait en réalité améliorer le C en ajoutant les outils qui, selon lui, manquaient pour se simplifier la vie.

Petit à petit, des collègues et d'autres passionnés commencent à s'intéresser au "C with classes". Et le besoin se fait sentir d'ajouter de nouvelles fonctionnalités. En 1983, les références, la surcharge d'opérateurs et les fonctions virtuelles sont ajoutées au langage qui s'appellera désormais C++. Le langage commence à ressembler un peu à ce que vous connaissez. En fait, quasiment tout ce que vous avez appris dans les parties I et II de ce cours est déjà présent dans le langage. Vous savez donc utiliser des notions qui ont de l'âge. 😊 Le C++ commence à intéresser les gens et Stroustrup finit par publier une version commerciale de son langage en 1985.

En 1989, la version 2.0 du C++ sort. Elle apporte en particulier l'héritage multiple, les classes abstraites et d'autres petites nouveautés pour les classes. Plus tard, les templates et les exceptions (deux notions que nous verrons dans la partie V de ce cours !) seront ajoutés au langage qui est quasiment équivalent à ce que l'on connaît aujourd'hui.

En parallèle à cette évolution, une bibliothèque plus ou moins standard se crée, dans un premier temps pour remplacer les "printf" et autres choses peu pratiques du C par les "cout" nettement plus faciles à utiliser. Cette partie de la SL, appelée *iostream* existe toujours et vous l'avez sûrement déjà utilisée. 😊

Plus tard d'autres éléments seront ajoutés à la bibliothèque standard, comme par exemple la STL, reprise de travaux plus anciens d'Alexander Stepanov notamment, et "traduits" de l'Ada vers le C++. Cet énorme ajout, qui va nous occuper dans la suite, est une avancée majeure pour le C++. C'est à ce moment-là que des choses très pratiques comme les `string` ou les `vector` apparaissent ! De nos jours, il est difficile d'imaginer un programme sans ces "briques" de base. Et je suis sûr que le reste de la



STL va aussi vous plaire. 😊

En 1998, un comité de normalisation se forme et décide de standardiser au niveau international le langage, c'est-à-dire que chaque implémentation du C++ devra fournir un certain nombre de fonctionnalités minimales. C'est là qu'est fixé le C++ actuel. C'est aussi à ce moment-là que la bibliothèque standard est figée et inscrite dans la norme. Cela veut dire que l'on devrait retrouver obligatoirement la bibliothèque standard avec n'importe quel compilateur. Et c'est ça qui fait sa force. Si vous avez un ordinateur avec un compilateur C++, il y aura forcément toutes les fonctions de la SL disponibles. Ce n'est pas forcément le cas d'autres bibliothèques qui n'existent que sous Windows ou que sous Linux par exemple.

Finalement, en 2011, le C++ devrait subir une révision majeure qui va lui apporter de nombreuses fonctionnalités attendues depuis longtemps. Parmi ces changements, on citera l'ajout de nouveaux mot-clés, la simplification des templates et l'ajout de nombreuses choses avancées à la bibliothèque standard (notamment des fonctionnalités venant de la célèbre bibliothèque `boost`).

Si la future norme, qu'on appelle C++1x, vous intéresse, je vous renvoie vers l'encyclopédie wikipédia: [C++1x sur Wikipédia](#).

Pourquoi une bibliothèque standard ?

C'est une question que beaucoup de monde pose. Sans la SL, le C++ ne contient en réalité pratiquement rien. Essayez d'écrire un programme sans `string`, `vector`, `cout` ou même `new` (qui utilise en interne des parties de la SL) ! C'est absolument impossible. Ou alors cela reviendrait à écrire nous-mêmes ces briques pour les utiliser par la suite, c'est-à-dire écrire notre propre version de `cout` ou de `vector`. Je vous assure que c'est très très compliqué à faire. 😞

Au lieu de réinventer la roue, les programmeurs se sont donc mis d'accord sur un ensemble de fonctionnalités de base utilisées par un maximum de personnes et les ont mises à disposition de tout le monde.

Tous les langages proposent des bibliothèques standards avec des éléments à disposition des utilisateurs. Le langage java, par exemple, propose même des outils pour faire des fenêtres alors que le C, lui, ne propose quasiment rien. Il faut donc souvent se fabriquer ses propres fonctions de base quand on utilise ce langage. Le C++ est un peu entre les deux puisque sa SL met à disposition des objets `vector` ou `string` mais ne propose pas de moyen de créer des fenêtres. C'est pour cela que nous avons appris à utiliser Qt. Il fallait utiliser quelque chose d'externe.

Bon ! Assez parlé. Voyons ce qu'elle a dans le ventre cette bibliothèque standard. 🤖

Le contenu de la SL

La bibliothèque standard est grosso-modo composée de trois grandes parties que nous allons explorer plus ou moins en détail dans ce cours. Comme vous allez le voir, il y a des morceaux que vous connaissez déjà bien (ou que vous devriez connaître 🧐).



Cette classification est assez arbitraire et selon les sources que l'on consulte on trouve jusqu'à 5 parties ou aucune séparation.

L'héritage du C

L'ensemble de la bibliothèque standard du C est présente dans la SL. Les 15 fichiers d'en-tête du C (norme de 1990) sont disponibles quasiment à l'identique en C++. De cette manière, les programmes écrits en C peuvent (presque tous) être réutilisés tels quels en C++. Je vous présenterai ces vénérables éléments venus du C à la fin de ce chapitre.

Les flux

Dans une deuxième partie, on trouve tout ce qui a trait aux flux, c'est-à-dire l'ensemble des outils permettant de faire communiquer les programmes avec l'extérieur. Ce sont les classes permettant :

- d'afficher des messages dans la console
- d'écrire du texte dans la console
- ou encore d'effectuer des opérations sur les fichiers

J'espère que cela vous rappelle quelque chose ! 😊

Cette partie sera brièvement abordée dans la suite de ce tutoriel, mais comme vous connaissez déjà bien ces choses, il ne sera

pas nécessaire de détailler tout ce qu'on trouve dans cette partie.

Nous avons déjà appris à utiliser `cout`, `cin` ainsi que les `fstream` pour communiquer avec des fichiers plus tôt dans ce cours. Je ne vais pas vous en apprendre beaucoup plus dans la suite. Mais nous allons quand même voir quelques fonctionnalités, comme la copie d'un fichier dans un tableau de mots. Vous verrez que certaines opérations fastidieuses peuvent être réalisées de manière très simple une fois que l'on connaît les bons concepts.

La STL

La *standard template library* (STL) est certainement la partie la plus intéressante. On y trouve des conteneurs, tels que les `vector`, permettant de stocker des objets selon différents critères. On y trouve également quelques algorithmes standards comme la recherche d'éléments dans un conteneur ou le tri d'éléments. On y trouve des itérateurs, des foncteurs, des prédicats, des pointeurs intelligents et encore plein d'autres choses mystérieuses que nous allons découvrir en détail.

Vous vous en doutez peut-être, la suite de ce cours sera consacrée principalement à la description de la STL.

Le reste

Je vous avais dit qu'il y avait trois parties... 🤔

Mais comme souvent, les classifications ne sont pas réellement utilisables dans la réalité. Il y a donc quelques éléments de la SL qui sont inclassables. En particulier la classe `string` qui est à la frontière entre la STL et les flux. De même certains outils concernant la gestion fine de la mémoire, les [paramètres régionaux](#) ou encore les [nombres complexes](#) ne rentrent dans aucune des trois parties principales. Mais cela ne veut pas dire que je ne vous en parlerai pas. 😊

Se documenter sur la SL

Dans le chapitre [Apprendre à lire la documentation de Qt](#), vous avez appris à vous servir d'une documentation pour trouver les classes et fonctions utiles... Et vous vous dites sûrement qu'il en est de même pour la bibliothèque standard. Je vais vous décevoir, mais ce n'est pas le cas. 😞

Il n'existe pas de "documentation officielle" pour la SL. Et c'est bien dommage.

En fait, ce n'est pas tout à fait vrai. La description très détaillée de chaque fonctionnalité se trouve dans la norme du langage. C'est un gros document en anglais de près de 800 pages absolument indigeste. Par exemple, on y trouve la description suivante pour les objets `vector` :

Citation : Norme C++, Paragraphe 23.2.4.1

A vector is a kind of sequence that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if `v` is a `vector<T, Allocator>` where `T` is some type other than `bool`, then it obeys the identity `&v[n] == &v[0] + n` for all $0 \leq n < v.size()$.



Même si vous parlez couramment anglais, vous n'avez certainement pas compris grand chose. Et pourtant, je vous ai choisi un passage pas trop obscur. Vous comprendrez donc que l'on ne peut pas travailler avec ça.

Des ressources sur le web

Heureusement, il existe quelques sites webs qui présentent le contenu de la SL. Mais manque de chance pour les anglophobes, toutes les références intéressantes sont en anglais. 🤔

Voici quatre sources que j'utilise régulièrement. Elles ne sont pas toutes complètes mais elles suffisent dans la plupart des cas. Je vous les ai classées de la plus simple à suivre à la plus compliquée.

- cplusplus.com. Une documentation plus simple qui présente l'ensemble de la SL. Les explications ne sont pas toujours complètes, mais elles sont accompagnées d'exemples simples qui permettent de bien comprendre l'intérêt de chaque fonction et classe.

- [Apache C++](#) Une bonne documentation de la SL en général. Les fonctions sont accompagnées d'exemples simples permettant de comprendre le fonctionnement de chacune d'elles. Elle est surtout intéressante pour sa partie sur les flux.
- [sgi](#). Une documentation très complète de la STL. La description n'est pas toujours aisée à lire pour un débutant et certaines choses présentées ne font pas partie de la STL standard mais seulement d'une version proposée par SGI. Présente uniquement la STL.
- [dinkumware.com](#). Une référence très complète et très bien faite sur la SL au complet. Le site présente également de la documentation sur TR1, la future bibliothèque standard du C++. Probablement la meilleure documentation sur la SL.

Je vous conseille de naviguer un peu sur ces sites et de regarder celui qui vous plait le plus.

L'autre solution est de me faire confiance et découvrir le tout via ce cours. Je ne pourrais bien sûr pas tout vous présenter, mais on va faire le tour de l'essentiel.

L'héritage du C

Le C++ étant en quelque sorte un descendant du C, la totalité de la bibliothèque standard du C est disponible dans la SL. Il y a quelques outils qui sont toujours utilisés, d'autres qui ont été remplacés par des versions améliorées et finalement d'autres qui sont totalement obsolètes. J'espère ne pas vous décevoir en ne parlant que des éléments utiles. C'est déjà beaucoup !



Si vous avez fait du C, vous devriez reconnaître les en-têtes dont je vais vous parler. La principale différence est le nom du fichier. En C, on utilise `math.h`, alors qu'en C++, c'est `cmath`. Le ".h" a disparu et un "c" a été ajouté devant le nom.

Comme tout le reste de la SL, la partie héritée du C est séparée en différents fichiers d'en-tête plus ou moins cohérents.

L'en-tête `cmath`

Celui-là, vous le connaissez déjà. Vous l'avez découvert tout au début du cours. C'est dans ce fichier que sont définies toutes les fonctions mathématiques usuelles. Comme je suis sympa, voici un petit rappel pour ceux qui dorment au fond de la classe.

Code : C++ - Rappel sur `cmath`

```
#include<iostream>
#include<cmath>
using namespace std;

int main()
{
    double a(4.3), b(5.2);
    cout << pow(a,b) << endl;    //Calcul de a^b
    cout << sqrt(a) << endl;    //Calcul de la racine carrée de a
    cout << cos(b) << endl;    //Calcul du cosinus de b
    return 0;
}
```

Ah je vois que vous vous en souvenez encore. Parfait ! C'est donc le fichier à inclure lorsque vous avez des calculs mathématiques à effectuer. Je ne vais pas vous réécrire toute la liste des fonctions, vous les connaissez déjà. 😊

Pour vous habituer à la documentation, essayez donc de retrouver ces fonctions dans les différentes ressources que je vous ai indiquées. Pour le site [cplusplus.com](#), vous devriez arriver sur cette [page](#).

L'en-tête `cctype`

Ce fichier propose quelques fonctions pour connaître la nature d'un `char`. Quand on manipule du texte, on doit souvent répondre à des questions comme :

- Cette lettre est-elle en majuscule ou en minuscule ?

- Ce caractère est-il un espace ?
- Ce symbole est-il un chiffre ?

Les fonctions présentes dans l'en-tête `cctype` sont là pour ça. Pour tester si un char donné est un chiffre, par exemple, on utilisera la fonction `isdigit()`. Comme dans l'exemple suivant :

Code : C++ - La fonction `isdigit()`

```
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    cout << "Entrez un caractere : ";
    char symbole;
    cin >> symbole;

    if(isdigit(symbole))
        cout << "C'est un chiffre." << endl;
    else
        cout << "Ce n'est pas un chiffre." << endl;

    return 0;
}
```

Comme vous le voyez, c'est vraiment très simple à utiliser. Le tableau suivant présente les fonctions les plus utilisées de cet en-tête. Vous trouverez la liste complète dans votre documentation favorite. 😊

Nom de la fonction	Description
<code>isalpha()</code>	Teste si le caractère est une lettre.
<code>isdigit()</code>	Teste si le caractère est un chiffre.
<code>islower()</code>	Teste si le caractère est une minuscule.
<code>isupper()</code>	Teste si le caractère est une majuscule.
<code>isspace()</code>	Teste si le caractère est un espace ou un retour à la ligne.

En plus de cela, il y a deux fonctions `tolower()` et `toupper()` qui convertissent une majuscule en minuscule et inversement.

On peut ainsi aisément transformer un texte en majuscule :

Code : C++ - Conversion d'une chaîne en majuscule

```
#include <iostream>
#include <cctype>
#include <string>
using namespace std;

int main()
{
    cout << "Entrez une phrase : " << endl;
    string phrase;
    getline(cin, phrase);

    //On parcourt la chaîne pour la convertir en majuscule
    for(int i(0); i<phrase.size(); ++i)
    {
        phrase[i] = toupper(phrase[i]);
    }
}
```

```
    }  
  
    cout << "Votre phrase en majuscule est : " << phrase << endl;  
    return 0;  
}
```

A nouveau, rien de bien sorcier. Je vous laisse vous amuser un peu avec ces fonctions. Essayez par exemple de réaliser un programme qui remplace tous les espaces d'une `string` par le symbole `#`. Je suis sûr que c'est dans vos cordes. 😊

L'en-tête `ctime`

Comme son nom l'indique, ce fichier d'en-tête contient plusieurs fonctions liées à la gestion du temps. La plupart sont assez bizarres à utiliser et donc peu utilisées. De toute façon, la plupart des autres bibliothèques, comme Qt, proposent des classes pour gérer les heures, les jours et les dates de manière plus aisée.

Personnellement, la seule fonction de `ctime` que j'utilise est la fonction `time()`. Elle renvoie le nombre de secondes qui se sont écoulées depuis le 1^{er} janvier 1970. 🤔 C'est ce qu'on appelle l'heure *UNIX*.

Code : C++ - L'heure UNIX

```
#include <iostream>  
#include <ctime>  
using namespace std;  
  
int main()  
{  
    int secondes = time(0);  
    cout << "Il s'est ecoule " << secondes << " secondes depuis le  
01/01/1970." << endl;  
    return 0;  
}
```



La fonction attend en argument un pointeur sur une variable dans laquelle stocker le résultat. Mais bizarrement, elle renvoie aussi ce résultat comme valeur de retour. L'argument est donc en quelque sorte inutile. On fournit généralement un pointeur ne pointant sur rien à la fonction. D'où le 0 passé en argument.

Ce qui donne :

Code : Console

```
Il s'est ecoule 1302471754 secondes depuis le 01/01/1970.
```

Ce qui fait beaucoup de secondes ! 🤔



Euh... A quoi ça sert ?


Il y a principalement trois raisons d'utiliser cette fonction :

- La première utilisation est bien sûr de calculer la date. Avec un petit peu d'arithmétique on retrouve facilement la date et l'heure actuelle. Mais comme je vous l'ai dit, la plupart des bibliothèques proposent des outils plus simples pour ça.
- Deuxièmement, on peut l'utiliser pour calculer le temps que met le programme pour s'exécuter. On appelle la fonction `time()` en début de programme puis une deuxième fois à la fin. Le temps passé dans le programme sera simplement la différence entre les deux valeurs obtenues !

- Le dernier cas d'utilisation de cette fonction vous l'avez déjà vu ! On l'utilise pour générer des nombres aléatoires. Nous allons voir comment dans la suite.

L'en-tête `cstdlib`

Voici à nouveau une vieille connaissance. Souvenez-vous, dans le [premier TP](#), je vous avais présenté un moyen de choisir un nombre au hasard. Il fallait utiliser les fonctions `srand()` et `rand()`.

C'est certainement l'en-tête le plus utile en C. Il contient toutes les briques de base et je crois qu'il n'y a pas un seul programme de C qui n'inclue pas `stdlib.h` (l'équivalent "C" de ce fichier). Par contre, en C++, ben... il ne sert quasiment à rien. Mise à part la génération de nombres aléatoires, tout a été remplacé par de nouvelles fonctionnalités en C++. 

Revoyons quand même en vitesse comment générer des nombres aléatoires. La fonction `rand()` renvoie un nombre au hasard entre 0 et `RAND_MAX` (un très grand nombre, généralement plus grand que 10^9). Si l'on souhaite obtenir un nombre au hasard entre 0 et 10, on utilise l'opérateur modulo (%).


Code : C++


```
nb = rand() % 10; //nb prendra une valeur au hasard entre 0 et 9
compris.
```

Jusque-là, rien de bien compliqué. Le problème est qu'un ordinateur ne sait pas générer un nombre au hasard. Tout ce qu'il sait faire c'est créer des suites de nombre qui *ont l'air aléatoires*. Il faut donc spécifier un début pour la séquence. Et c'est là qu'intervient la fonction `srand()`. Elle permet de spécifier le premier terme de la suite.



Il ne faut appeler qu'une seule et unique fois la fonction `srand()` par programme !

Le problème est que si l'on donne chaque fois le même premier terme à l'ordinateur, il va nous générer à chaque fois la même séquence ! Il faut donc lui donner quelque chose de différent à chaque exécution du programme. Et qu'est-ce qui change à chaque fois que l'on exécute un programme ?  La date et l'heure bien sûr !

La bonne solution est donc d'utiliser le résultat de la fonction `time()` comme premier terme de la série. 


Code : C++ - Générer des nombres au hasard

```
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

int main()
{
    srand(time(0)); //On initialise la suite de nombres
    aléatoires

    for(int i(0); i<10; ++i)
        cout << rand() % 10 << endl; //On génère des nombres au
    hasard

    return 0;
}
```

Libre à vous ensuite d'utiliser ces nombres pour mélanger des lettres comme dans le TP ou pour créer un jeu de casino. Le principe de base est toujours le même. 

Les autres en-têtes

Mis à part `cassert` dont nous parlerons plus tard, le reste des 15 en-têtes du C ne sont que très rarement utilisés en C++. Je ne vous en dirai donc pas plus dans ce cours.

Bon, assez travaillé avec les reliques du C ! Pour l'instant, je ne vous ai pas présenté de grandes révolutions pour vos programmes comme je vous l'avais promis. Ne le dites pas trop fort si vous rencontrez des amateurs de C, mais c'est parce qu'on n'a pas encore utilisé la puissance du C++. Attachez vos ceintures, la suite du voyage va secouer.
Bon, bon, bon, c'est bien joli tout ça, mais c'est quand qu'on programme ?

On y vient ne vous en faites pas. 😊

Il est temps d'explorer tout ça !

Les conteneurs

Après cette brève introduction à la SL et aux éléments venus du C, il est temps de se plonger dans ce qui fait la force de la bibliothèque standard, la fameuse **STL**.

Ce sigle signifie *Standard Template Library*, que l'on pourrait traduire par "Bibliothèque standard basée sur des templates". Pour l'instant, vous ne savez pas ce que sont les *templates*, nous les découvrirons plus tard. Mais cela ne veut pas dire que vous n'avez pas le niveau requis ! Souvenez-vous de la classe `string`, vous avez appris à l'utiliser bien avant de savoir ce qu'était un objet. Il en sera de même ici, nous allons utiliser (beaucoup) de templates sans que vous ayez besoin d'en savoir plus à leur sujet. 😊

L'élément de base autour duquel toute la STL est basée est le conteneur. Ce sont des objets permettant de stocker d'autres objets. D'ailleurs, vous en connaissez déjà un : le `vector`. Dans ce premier vrai chapitre sur la SL, vous allez découvrir qu'il existe d'autres sortes de conteneur pour tous les usages.

La vraie difficulté sera alors de faire son choix parmi tous ces conteneurs. Mais ne vous en faites pas, je serai là pour vous guider. 😊

Stocker des éléments

Vous l'avez vu tout au long de ce cours, stocker des objets dans d'autres objets est une opération très courante. Pensez par exemple aux collections hétérogènes lorsque nous avons vu le polymorphisme ou aux différents layouts dans Qt qui permettaient d'arranger les boutons et autres objets dans les widgets. Ces moyens de stockage s'appellent des **conteneurs** en langage informatique. Ce sont des objets qui peuvent contenir toute une série d'autres objets et qui proposent des méthodes permettant de les manipuler.

A priori, cette définition peut faire un peu peur. 😊

Mais ce ne devrait pas être le cas. Cela fait bien longtemps que vous avez appris à utiliser les `vector`, le membre le plus connu de la STL. Voici un petit rappel basique pour ceux qui dormaient au fond de la salle de cours. 😊

Code : C++ - Petit rappel sur `vector`

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> tab(5,4);    //Un tableau contenant 5 entiers dont
    la valeur est 4
    tab.pop_back();        //On supprime la dernière case du
    tableau.
    tab.push_back(6);       //On ajoute un 6 à la fin du tableau.

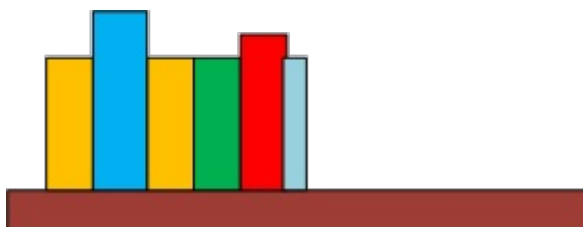
    for(int i(0); i<tab.size(); ++i) //On utilise size() pour
    connaitre le nombre d'éléments dans le vector
        cout << tab[i] << endl;    //On utilise les crochets [] pour
    accéder aux éléments

    return 0;
}
```



Souvenez-vous, la première case d'un `vector` possède toujours l'indice 0.

Les `vector` sont des tableaux dynamiques. Autrement dit, les éléments qu'ils contiennent sont stockés les uns à côté des autres dans la mémoire. On pourrait se dire que c'est la seule manière de ranger des objets. En tout cas, c'est comme ça que la plupart des gens rangent leurs caves ou leurs étagères. Je suis sûr que vous faites de même. 😊



Cette manière de ranger des livres sur une bibliothèque est sans doute la plus simple que l'on puisse imaginer. On peut accéder directement au 3^{ème} ou au 8^{ème} livre en tendant simplement le bras. Et comme on le voit sur l'illustration, ajouter des livres à droite de la collection est aussi très rapide : il suffit de les poser à côté des autres.

Mais pour d'autres opérations, cette méthode de rangement n'est pas forcément la meilleure. Si vous devez ajouter un livre au milieu de la collection (entre le vert et le rouge sur l'image), vous allez devoir décaler tous ceux situés à droite. Ici, ce n'est pas un gros travail. Mais imaginez que votre bibliothèque contienne des centaines de livres, tout décaler va prendre du temps. De même, ôter un livre au milieu de l'étagère va être coûteux, il va à nouveau falloir tout déplacer !

Ce ne sont pas les seules opérations difficiles à effectuer avec des livres. Trier les livres selon le nom de l'auteur est aussi quelque chose de long et difficile à réaliser. Si le tri avait été effectué au moment où les livres ont été posés pour la première fois, on n'aurait plus à le faire. Par contre, ajouter un livre dans la collection implique une réflexion préalable. Il faut, en effet, placer le bouquin au bon endroit pour que la collection reste triée. 🤔

Inverser l'ordre des livres est aussi un long travail dans une grande bibliothèque. Bref, ranger des objets n'est pas aussi simple qu'on pourrait le penser.

Vous l'avez sûrement constaté, toutes les bibliothèques rangent leurs livres les uns à côté des autres. Mais les informaticiens sont des gens malins. Ils ont inventé d'autres méthodes de rangement. Nous allons les découvrir à partir de maintenant.



Il n'existe pas de "conteneur ultime", pour lequel toutes les opérations sont rapides. Il faut choisir la méthode de stockage adaptée à chaque problème en fonction des opérations que l'on veut privilégier. Je vous donnerai quelques astuces à la fin de ce chapitre.

Les deux catégories de conteneurs

Les différents conteneurs peuvent être partagés en deux catégories selon que les éléments sont classés à la suite les uns des autres ou pas. On parle alors de séquences et de conteneurs associatifs. Les `vector` sont bien évidemment des séquences puisque comme je vous l'ai dit, toutes les cases sont arrangées de manière contiguë dans la mémoire.

Nous allons voir tous ces conteneurs en détail. Pour l'instant, voici un tableau contenant tous les conteneurs de la STL selon leur catégorie.

Séquences	Conteneurs associatifs
<code>vector</code> <code>deque</code> <code>list</code> <code>stack</code> <code>queue</code> <code>priority_queue</code>	<code>set</code> <code>multiset</code> <code>map</code> <code>multimap</code>



Les noms des conteneurs sont en anglais et peut-être un peu bizarres, mais vous allez vite vous y faire. Et puis, les noms, bien que compliqués, décrivent plutôt bien à quoi ils servent.



Pour utiliser ces conteneurs, il faut inclure le fichier d'en-tête correspondant. Et là rien de bien sorcier. Pour utiliser des `list`, il faut ajouter la ligne `#include <list>` en haut de votre code. De même pour utiliser une `map`, c'est `#include <map>` en début de fichier qui fera votre bonheur.

Vous vous dites peut-être qu'apprendre à utiliser 15 conteneurs différents va demander beaucoup de travail. Je vous rassure tout de suite, ils sont quand même très similaires. Après tout, ils sont tous là pour stocker des objets ! Et comme les concepteurs de la

STL sont sympas, ils ont donnés les mêmes noms aux méthodes communes de tous les conteneurs. 🤔

Par exemple, la méthode `size()` renvoie la taille d'un `vector`, d'une `list` ou d'une `map`. Magique ! 🧙

Quelques méthodes communes

Connaître la taille, c'est bien, mais on a parfois juste besoin de savoir si le conteneur est vide ou pas. Pour cela, il existe la méthode `empty()` qui renvoie **true** si le conteneur est vide et **false** sinon.

Code : C++

```
list<double> a; //Une liste de double
if(a.empty())
    cout << "La liste est vide." << endl;
else
    cout << "La liste n'est pas vide." << endl;
```

Vous ne savez pas encore ce que sont les listes ni comment et quand les utiliser mais je crois que vous n'avez pas eu de peine à comprendre cet extrait de code. 😊

Une autre méthode bien pratique est celle qui permet de vider entièrement un conteneur. Il s'agit de `clear()`. Cela ne devrait pas surprendre les anglophones parmi vous !

Code : C++

```
set<string> a; //Un ensemble de chaines de caractères
//Quelques actions...
a.clear(); //Et on vide le tout !
```

A nouveau, rien de bien difficile, même avec une classe dont vous ne savez rien.

Finalement, il arrive qu'on ait besoin d'échanger le contenu de deux conteneurs de même type. Et plutôt que de devoir copier les éléments un à un à la main, les concepteurs de la STL ont créé une méthode `swap()` qui effectue cet échange de la manière la plus efficace possible.

Code : C++

```
vector<double> a(8,3.14); //Un vector contenant 8 fois le nombre
3.14
vector<double> b(5,2.71); //Un autre vector contenant 5 fois le
nombre 2.71

a.swap(b); //On échange le contenu des deux tableaux.
//b a maintenant une taille de 8 et a une taille de 5.
```



Comme nous le verrons dans la partie suivante du cours, `vector<int>` et `vector<double>` sont des types différents. On ne peut donc pas échanger le contenu de deux conteneurs dont les éléments sont de type différent.

Bon bon, moi ça m'a donné envie d'en savoir plus sur ces conteneurs. Tournons nous donc vers les séquences. 😊

Les séquences et leurs adaptateurs

Commençons avec notre vieil ami, le `vector`.

Les vector, encore et toujours

Si vous parlez la langue de Shakespeare, vous aurez certainement reconnu dans le nom de ces objets, le mot "vecteur", ces drôles d'objets mathématiques que l'on représente par des flèches. Eh bien, ils n'ont pas énormément de choses en commun ! 😊

Les `vector` ne sont vraiment pas adaptés pour effectuer des opérations mathématiques. Et en plus, ils n'en ont même pas les caractéristiques. On pourrait dire que c'est un mauvais choix de nom de la part des concepteurs de la STL. Mais bon, il est trop tard pour changer... Vous allez donc devoir vous habituer à ce faux-ami. 😊

Comme vous l'avez vu depuis longtemps, les `vector` sont très simples à utiliser. On accède aux éléments via les crochets `[]`, comme pour les tableaux statiques et l'ajout d'éléments à la fin se fait via la méthode `push_back()`. En réalité cette méthode est une opération commune à toutes les séquences. Il en est de même pour `pop_back()`.

Il existe en plus de ça deux méthodes plus rarement utilisées permettant d'accéder au premier et au dernier élément d'un `vector` ou de toute autre séquence. Il s'agit des méthodes `front()` et `back()`. Mais comme il n'est que peu souvent utile d'accéder qu'au premier ou qu'au dernier élément, ces méthodes ne présentent que peu d'intérêt.

Finalement, il existe la méthode `assign()` permettant de remplir tous les éléments d'une séquence avec la même valeur.

Récapitulons tout ça dans un tableau.


Méthode	Description	Mini exemple
<code>push_back()</code>	Ajout d'un élément à la fin du tableau.	Code : C++ <pre>vector<int> a(5,3); //Un vector de 5 entiers valant 3 a.push_back(4); //Ajout d'une case avec le nombre 4 à la fin du tableau</pre>
<code>pop_back()</code>	Suppression de la dernière case du tableau.	Code : C++ <pre>vector<int> a(5,3); //Un vector de 5 entiers valant 3 a.pop_back(); //Suppression de la dernière case</pre>
<code>front()</code>	Accès à la première case du tableau.	Code : C++ <pre>vector<int> a(5,3); //Un vector de 5 entiers valant 3 a.front() = 4; //Le premier élément du tableau vaut maintenant 4</pre>
<code>back()</code>	Accès à la dernière case du tableau.	Code : C++ <pre>vector<int> a(5,3); //Un vector de 5 entiers valant 3 a.back() = 4; //Le dernier élément du tableau vaut maintenant 4</pre>
<code>assign()</code>	Modifie le contenu d'un tableau.	Code : C++ <pre>vector<int> a(5,3); //Un vector de 5 entiers valant 3</pre>

tableau.

```
a.assign(6,2); //Le tableau contient
maintenant 6 fois le nombre 2.
```



Si vous appelez `assign(4,5)` sur un `vector` de 8 éléments, seuls les 4 premiers seront modifiés. Le tableau sera donc plus petit.

En plus des crochets, il est possible d'accéder aux éléments d'un `vector` en utilisant des *itérateurs*. C'est ce que nous allons découvrir dans le prochain chapitre. 

Pour l'instant, tournons-nous vers les autres types de séquences.

Les deque, ces drôles de tableaux

"deque" est en fait un acronyme (bizarre) pour *double ended queue*, ce qui donne en français, "queue à deux bouts". Derrière ce nom un peu original se cache un concept très simple : c'est un tableau auquel on peut ajouter des éléments aux deux extrémités. Les `vector` proposent les méthodes `push_back()` et `pop_back()` pour manipuler ce qui se trouve à la fin du tableau. Modifier ce qui se trouve au début n'est pas possible. Les deque lèvent cette limitation en proposant des méthodes `push_front()` et `pop_front()`. Elles sont aussi très simples à utiliser. La seule difficulté vient du fait que le premier élément possède toujours l'indice 0. Les indices sont donc décalés à chaque ajout en début de deque.

Code : C++ - Ajout d'éléments au début d'une deque

```
#include <deque> //Ne pas oublier !
#include <iostream>
using namespace std;

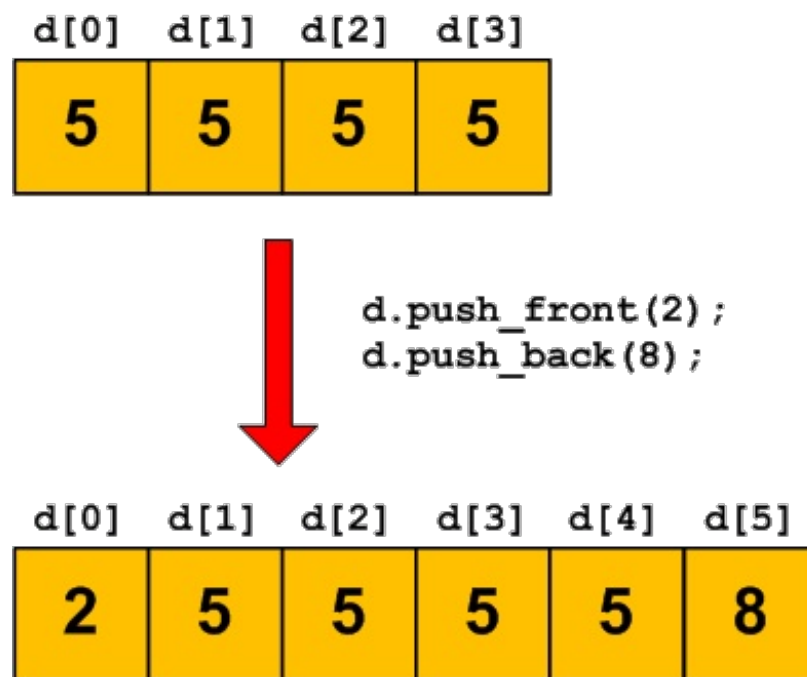
int main()
{
    deque<int> d(4,5); //Une deque de 4 entiers valant 5

    d.push_front(2);    //On ajoute le nombre 2 au début
    d.push_back(8);     //Et le nombre 8 à la fin

    for(int i(0); i<d.size(); ++i)
        cout << d[i] << " ";    //Affiche 2 5 5 5 5 8

    return 0;
}
```

Et pour bien comprendre le tout, je vous propose un petit schéma :



Même si l'on ajoute des éléments au début d'une deque, l'indice du premier élément est toujours le 0. Je vous l'ai déjà dit, mais je préfère vous éviter des soucis avec votre compilateur. 😊

Bon, je crois que vous avez compris. 😊 Si vous avez survécu aux premiers chapitres de ce cours, tout ça doit vous sembler bien facile.

Les stack, une histoire de pile

La classe `stack` est la première structure de donnée un peu spéciale que vous rencontrez. C'est un conteneur qui n'autorise l'accès qu'au dernier élément ajouté. 😊

En fait, il n'y a que 3 opérations autorisées :

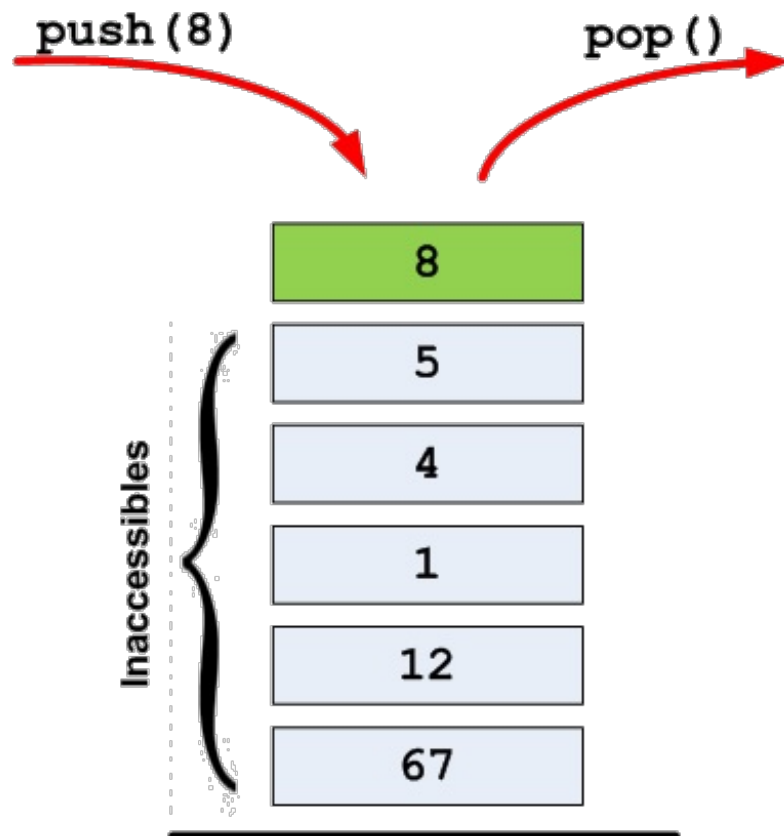
1. Ajouter un élément.
2. Consulter le dernier élément ajouté.
3. Supprimer le dernier élément ajouté.

Cela se fait via les trois méthodes `push()`, `top()` et `pop()`.



Je ne comprends pas bien l'intérêt d'un tel stockage !

En termes techniques, on parle de *structure LIFO*. Le dernier élément ajouté est le premier à pouvoir être ôté. Comme sur une pile d'assiette ! Vous ne pouvez accéder qu'à la dernière assiette posée sur la pile.



Cela permet d'effectuer des traitements sur les données en ordre inverse de leur arrivée dans la pile. Comme pour les assiettes. La dernière assiette sale sur la pile est la première à être lavée. Alors que celle arrivée en premier (et qui est donc tout en-bas de la pile) sera traitée en dernier. 🤔

Un exemple plus informatique serait la gestion d'un stock. On ajoute le nombre d'articles vendus chaque mois à notre pile et on consulte les trois derniers ajouts sans s'occuper du reste pour créer le bilan trimestriel.

Code : C++ - Utilisation d'une pile

```
#include <stack>
#include <iostream>
using namespace std;

int main()
{
    stack<int> pile;           //Une pile vide
    pile.push(3);             //On ajoute le nombre 3 à la pile
    pile.push(4);
    pile.push(5);

    cout << pile.top() << endl; //On consulte le sommet de la pile
    (le nombre 5)

    pile.pop();               //On supprime le dernier élément ajouté (le
    nombre 5)

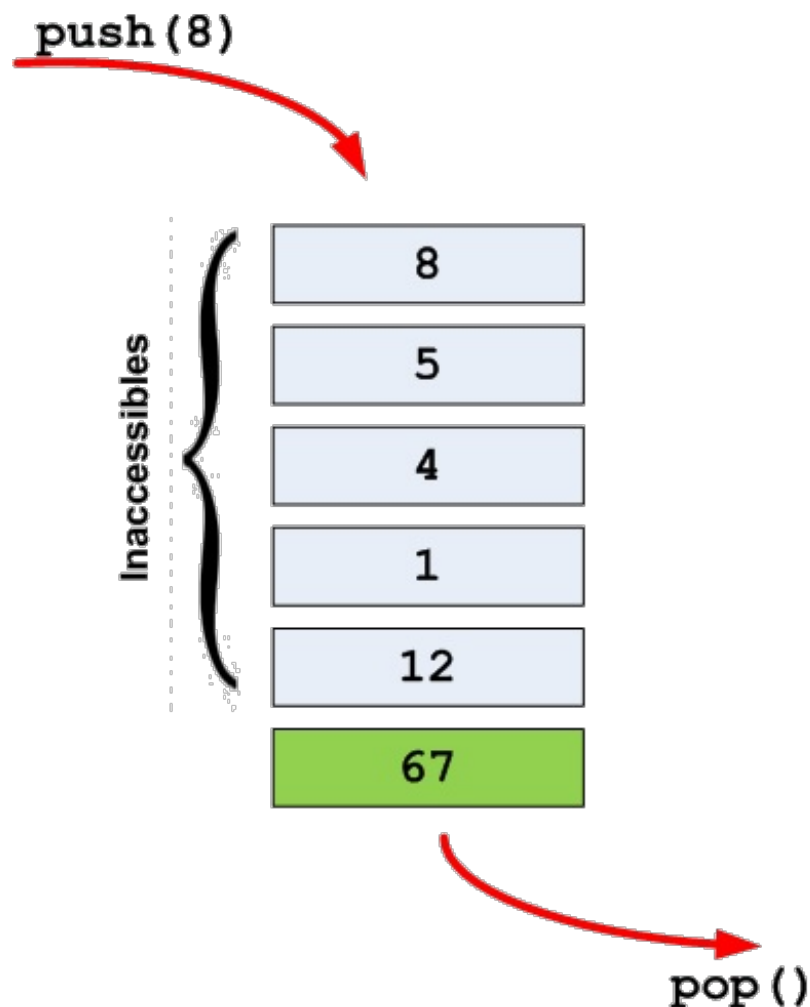
    cout << pile.top() << endl; //On consulte le sommet de la pile
    (le nombre 4)

    return 0;
}
```

Peut-être que vous aurez un jour besoin de ce genre de structures. Repensez alors à ce chapitre !

Les queue, une histoire de file

Les files sont très similaires aux piles (et pas que pour leurs noms !). En termes techniques, on parle de *structure FIFO*. La différence ici est que l'on ne peut accéder qu'au premier élément ajouté. Exactement comme dans une file de supermarché. Les gens attendent les uns derrière les autres et la caissière traite les courses de la première personne arrivée. Quand elle a terminé, elle s'occupe de la deuxième et ainsi de suite.



Le fonctionnement est identique à celui des piles. La seule différence est qu'on utilise `front()` pour accéder à ce qui se trouve à l'avant de la file au lieu de `top()`.

Les priority_queue, la fin de l'égalité

Les `priority_queue` sont des queue qui ordonnent leurs éléments. Un peu comme si les clients avec les plus gros paquets de course passaient avant les gens avec seulement un ou deux articles. Les méthodes sont exactement les mêmes que dans le cas des files simples.

Code : C++ - Utilisation d'une priority_queue

```
#include <queue> //Attention ! queue et priority_queue sont définies
dans le même fichier
#include <iostream>
using namespace std;

int main()
{
    priority_queue<int> file;
    file.push(5);
```

```

        file.push(8);
        file.push(3);

        cout << file.top() << endl; //Affiche le plus grand des
        éléments insérés (le nombre 8)

        return 0;
    }

```



Les objets stockés dans une `priority_queue` doivent avoir un opérateur de comparaison (<) surchargé afin de pouvoir être classés !

On utilise par exemple ce genre de structure pour gérer des événements selon leur priorité. Pensez aux signaux et slots de Qt. On pourrait leur affecter une valeur pour traiter les événements dans un certain ordre.

Les list, à voir plus tard

Finalement, le dernier conteneur sous forme de séquence est la liste. Cependant, pour les utiliser de manière efficace il faut savoir manipuler les itérateurs, ce que nous apprendrons à faire dans le prochain chapitre. De toute façon, je crois que je vous ai assez parlé de séquences pour le moment. Il est temps de parler d'une tout autre manière de ranger des objets. 🤔

Les tables associatives

Jusqu'à maintenant, vous êtes habitués à accéder aux éléments d'un conteneur en utilisant les crochets []. Dans un `vector` ou une `deque`, les éléments sont accessibles via leur index, un nombre entier positif. Ce n'est pas toujours très pratique. Imaginez un dictionnaire, vous n'avez pas besoin de savoir que "banane" est le 832^{ème} mot pour accéder à sa définition. Les tables associatives sont des structures de données qui autorisent l'emploi de n'importe quel type comme index. En termes techniques, on dit qu'une `map` est une table associative permettant de stocker des paires clé-valeur.

Concrètement, cela veut dire que vous pouvez créer un conteneur où les indices sont des `string` par exemple. Comme le type des indices peut varier, il faut l'indiquer lors de la déclaration de l'objet :

Code : C++

```

#include <map>
#include <string>
using namespace std;

map<string, int> a;

```

Ce code déclare une table associative qui stocke des entiers mais dont les indices sont des chaînes de caractères. On peut alors accéder à un élément via les crochets [] comme ceci :

Code : C++

```

a["salut"] = 3; //La case "salut" de la map vaut maintenant 3

```

Si la case n'existe pas, elle est automatiquement créée.

On peut utiliser ce que l'on veut comme clé. La seule condition est que l'objet utilisé possède un opérateur de comparaison "plus-petit-que" (<).

Avec ce nouvel outil, on peut très facilement compter le nombre d'occurrences d'un mot dans un fichier. Essayez par vous-même c'est un très bon exercice. Le principe est simple. On parcourt le fichier et pour chaque mot on incrémente la case correspondante dans la table associative. Voici ma solution :

Code : C++

```
#include <map>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    ifstream fichier("texte.txt");
    string mot;
    map<string, int> occurrences;
    while(fichier >> mot)    //On lit le fichier mot par mot
    {
        ++occurrences[mot]; //On incrémente le compteur pour le
mot lu
    }
    cout << "Le mot 'banane' est present " << occurrences["banane"]
<< " fois dans le fichier" << endl;
    return 0;
}
```

On peut difficilement faire plus court ! Pour le moment en tout cas. 😊

Les map ont un autre gros point fort : les éléments sont triés selon leur clé. Donc si l'on parcourt une map du début à la fin, on parcourt les clés de la plus petite à la plus grande. Le problème, c'est que pour parcourir une table associative du début à la fin, il faut utiliser les itérateurs et donc attendre le prochain chapitre. 😊

Les autres tables associatives

Les autres tables sont des variations de la map. Le principe de fonctionnement de ces conteneurs est très similaire, mais à nouveau, il nous faut utiliser les itérateurs pour exploiter la pleine puissance de ces structures de données. Je sens que vous allez bientôt avoir envie d'en savoir plus sur ces drôles de bêtes...

En attendant, je vais quand même vous en dire quelques mots sur ces autres structures de données. 😊

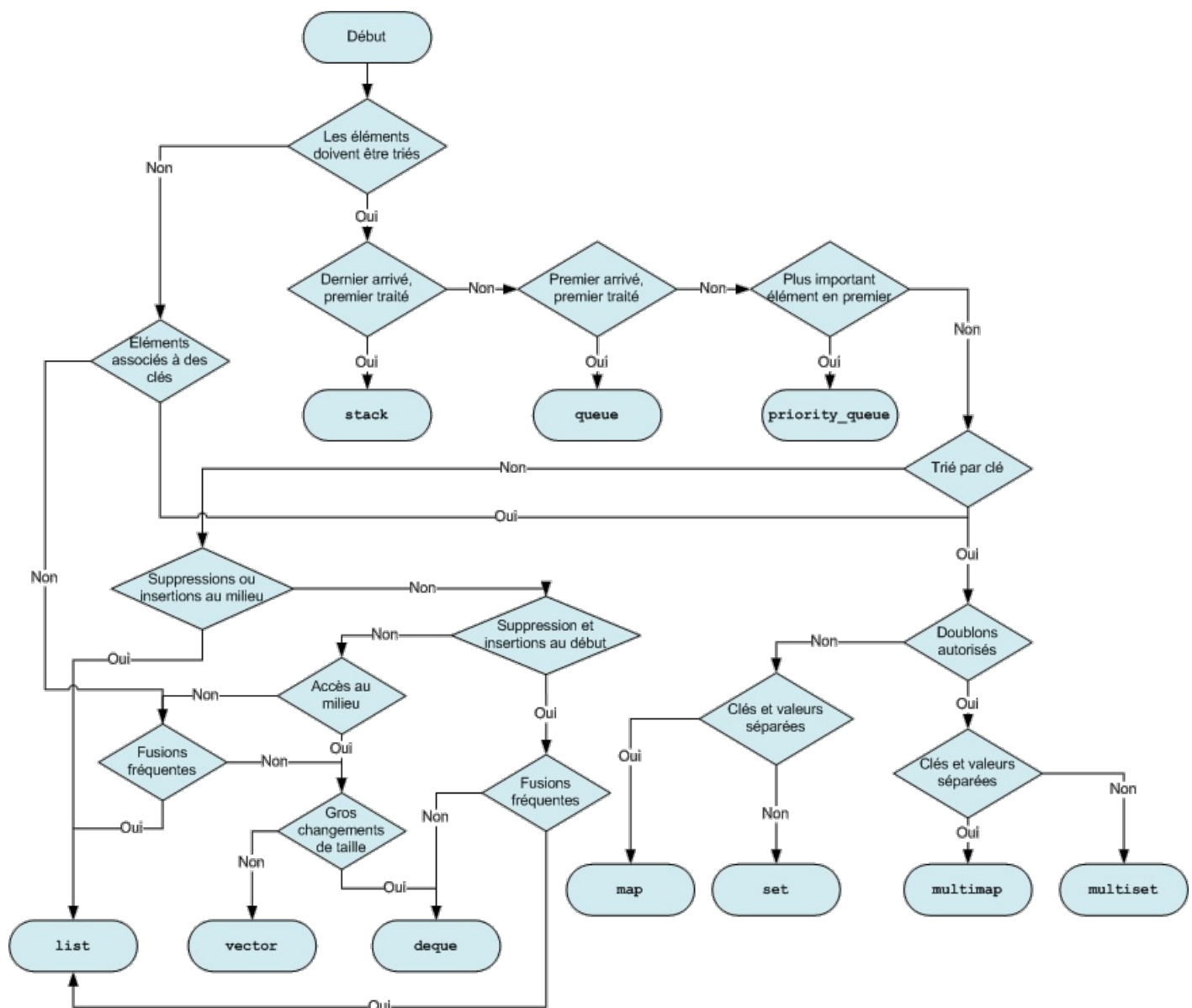
- Les `set` sont utilisés pour représenter les ensembles. On peut insérer des objets dans l'ensemble et y accéder via une méthode de recherche. Par contre, il n'est pas possible d'y accéder via les crochets. En fait, c'est comme si on avait une map où les clés et les éléments étaient confondus.
- Les `multiset` et `multimap` sont des copies des `set` et `map` où chaque clé peut exister en plusieurs exemplaires.

On reparlera un peu de tout cela, mais ces trois derniers conteneurs sont quand même d'un usage plus rare. 😊

Choisir le bon conteneur

La principale difficulté avec la STL est de choisir le bon conteneur ! Comme dans l'exemple de la bibliothèque de livres, faire le mauvais choix peut avoir des conséquences désastreuses en terme de performances. Et puis, tous les conteneurs n'offrent pas les mêmes services. Avez-vous besoin d'accéder aux éléments directement ? Ou préférez-vous les trier et n'accéder qu'à l'élément avec la plus grande priorité ? C'est à ce genre de questions qu'il faut répondre pour faire le bon choix. Et ce n'est pas facile ! 😊

Heureusement, je vais vous aider via un schéma. En suivant les flèches et en répondant aux questions posées dans les losanges, on tombe sur le conteneur le plus approprié.



Avec ça, pas moyen de se tromper ! 🤪

Il est évidemment inutile d'apprendre ce schéma par cœur. Sachez simplement qu'il existe et où le trouver.

Au final, on utilise souvent des `vector`. Cet outil de base permet de résoudre bien des problèmes sans trop se poser de questions. Et on sort une `map` quand on a besoin d'utiliser autre chose que des entiers pour indexer les éléments. Utiliser ce schéma c'est le niveau supérieur, mais choisir le bon conteneur peut devenir essentiel quand on cherche à créer un programme vraiment optimisé.

Bon, récapitulons. Vous savez maintenant choisir le conteneur adapté à vos besoins et vous savez le remplir. C'est bien, mais vous en conviendrez on ne va pas très loin avec juste ces notions.

La prochaine étape est bien sûr d'apprendre à parcourir ces conteneurs pour utiliser les objets qui y sont stockés. Ça vous tente ? Parfait ! Alors ne quittez pas votre clavier, c'est ce que je vais vous présenter dans le chapitre suivant.

Itérateurs et foncteurs

Dans le chapitre précédent, vous avez pu vous familiariser un peu avec les différents conteneurs de la STL.

Vous avez appris à ajouter des éléments à l'intérieur, mais vous n'avez guère fait plus excitant. Vous avez dû rester un peu sur votre faim. Il faut bien sûr apprendre à parcourir les conteneurs et à appliquer des traitements aux éléments. Pour ce faire : nous allons avoir besoin de deux notions, les *itérateurs* et les *foncteurs*. 😊

Les itérateurs sont des objets ressemblant aux pointeurs qui vont nous permettre de parcourir les conteneurs. L'intérêt de ces objets est qu'on les utilise de la même manière quel que soit le conteneur ! Pas besoin de faire de distinction entre les `vector`, les `map` ou les `list`. Vous allez voir, c'est magique.

Les foncteurs, quant à eux, sont des objets que l'on va utiliser comme fonction. Nous allons alors pouvoir appliquer ces fonctions à tous les éléments d'un conteneur par exemple.

Prenez une grande inspiration, ce chapitre va changer beaucoup de choses dans votre vie ! 😊

Itérateurs : des pointeurs boostés

Dans les premiers chapitres de ce cours, nous avons vu que les pointeurs pouvaient être assimilés à des flèches pointant sur les cases de la mémoire de l'ordinateur. Ce n'est bien sûr qu'une image, mais elle va nous aider dans la suite. 😊

Un conteneur est un objet contenant des éléments. Un peu comme la mémoire contient des variables. Les concepteurs de la STL ont donc eu l'idée de créer des pointeurs spéciaux pour se déplacer dans les conteneurs comme le ferait un pointeur dans la mémoire. Ces pointeurs spéciaux s'appellent des *itérateurs*.



Les itérateurs sont en réalité des objets plutôt complexes. Pas juste de simples pointeurs.

L'avantage de cette manière de faire est qu'elle réutilise quelque chose que l'on connaît bien. On peut déplacer l'itérateur en utilisant les opérateurs `++` et `--`, comme on pourrait le faire pour un pointeur. Mais l'analogie ne s'arrête pas là, on accède à l'élément pointé (ou itéré 😊) via l'étoile `*`. Bref, ça nous rappelle de vieux souvenirs. Du moins j'espère...

Déclarer un itérateur...

Chaque conteneur possède son propre type d'itérateur, mais la manière de les déclarer est toujours la même. Comme toujours, il faut un type et un nom. Choisir un nom, c'est votre problème 😊, mais pour le type, je vais vous aider. Il faut mettre le type du conteneur, suivi de l'opérateur `::` et du mot `iterator`. Par exemple pour un itérateur sur un `vector` d'entiers, on a :

Code : C++ - Itérateur sur un vector d'entiers

```
#include <vector>
using namespace std;

vector<int> tableau(5,4);           //Un tableau de 5 entiers valant 4
vector<int>::iterator it;          //Un itérateur sur un vector d'entiers
```

Voici encore quelques exemples :

Code : C++ - D'autres itérateurs

```
map<string, int>::iterator it1;    //Un itérateur sur les tables
associatives string-int

deque<char>::iterator it2;         //Un itérateur sur une deque de
caractères

list<double>::iterator it3;        //Un itérateur sur une liste de
nombres à virgule
```

Bon. Je crois que vous avez compris. 😊

... et itérer

Il ne nous reste plus qu'à les utiliser. Tous les conteneurs possèdent une méthode `begin()` renvoyant un itérateur sur le premier élément contenu. On peut ainsi faire pointer l'itérateur sur le premier élément. On avance alors dans le conteneur en utilisant l'opérateur `++`. Il ne nous reste plus qu'à spécifier une condition d'arrêt. On ne veut pas aller en-dehors du conteneur. Pour ce faire, les conteneurs possèdent une méthode `end()` renvoyant un itérateur sur la fin du conteneur.



En réalité, `end()` renvoie un itérateur sur un élément en-dehors du conteneur. Il faut donc itérer jusqu'à `end()` non compris.

On peut donc parcourir un conteneur en itérant dessus depuis `begin()` jusqu'à `end()`. Voyons ça avec un exemple :

Code : C++ - Itérer sur une deque

```
#include<deque>
#include <iostream>
using namespace std;

int main()
{
    deque<int> d(5,6);           //Une deque de 5 éléments valant 6
    deque<int>::iterator it;    //Un itérateur sur une deque d'entiers

    //Et on itère sur la deque
    for(it = d.begin(); it!=d.end(); ++it)
    {
        cout << *it << endl;    //On accède à l'élément pointé via
        l'étoile
    }
    return 0;
}
```



Les itérateurs ne sont pas optimisés pour l'opérateur de comparaison. On ne devrait donc pas écrire `it<d.end()` comme on en a l'habitude avec les index de tableau. Utiliser `!=` est plus efficace.

Simple non ? Si vous avez aimé les pointeurs (😬), vous allez adorer les itérateurs. Pour les `vector` et les `deque`, cela peut vous sembler inutile. On peut faire aussi bien avec les crochets `[]`. Mais pour les `map` et surtout les `list`, ce n'est pas vrai, les itérateurs sont le seul moyen que nous avons de les parcourir.

Des méthodes uniquement pour les itérateurs

Même pour les `vector` ou `deque`, il existe des méthodes qui nécessitent l'emploi d'itérateurs. Il s'agit en particulier des méthodes `insert()` et `erase()` qui, comme leur nom l'indique, permettent d'ajouter ou supprimer un élément au milieu d'un conteneur. Jusqu'à maintenant, vous ne pouviez qu'ajouter des éléments à la fin d'un conteneur, jamais au milieu. La raison est simple : pour ajouter quelque chose au milieu, il faut indiquer où l'on souhaite insérer l'élément. Et ça, c'est justement le but d'un itérateur.

Un exemple vaut mieux qu'un long discours.

Code : C++

```
#include <vector>
#include <string>
#include <iostream>
```

```
using namespace std;

int main()
{
    vector<string> tab;    //Un tableau de mots

    tab.push_back("les"); //On ajoute deux mots dans le tableau
    tab.push_back("Zeros");

    tab.insert(tab.begin(), "Salut"); //On insère le mot "Salut" au
    début

    //Affiche les mots donc la chaine "Salut les Zeros"
    for(vector<string>::iterator it=tab.begin(); it!=tab.end();
    ++it)
    {
        cout << *it << " ";
    }

    tab.erase(tab.begin()); //On supprime le premier mot

    //Affiche les mots donc la chaine "les Zeros"
    for(vector<string>::iterator it=tab.begin(); it!=tab.end();
    ++it)
    {
        cout << *it << " ";
    }

    return 0;
}
```

Et c'est la même chose pour tous les types de conteneurs. Si vous avez un itérateur sur un élément, vous pouvez le supprimer via `erase()` ou ajouter un élément juste après grâce à `insert()`.



Souvenez-vous quand même que les `vector` ne sont pas optimisés pour l'insertion et l'effacement au milieu. Le schéma du chapitre précédent vous aidera à faire un meilleur choix si vous avez vraiment besoin de faire ce genre de modifications sur votre conteneur.

Je vous avais dit que vous alliez adorer ce chapitre ! Et ça ne fait que commencer. 😊

Les différents itérateurs

Terminons quand même avec quelques aspects un petit peu plus techniques. Il existe en réalité 5 sortes d'itérateurs. Lorsque l'on déclare un `vector::iterator` ou un `map::iterator`, on déclare en réalité un objet d'une de ces cinq catégories. Cela se fait via une redéfinition de type, chose que nous verrons dans la cinquième partie de ce cours.

Parmi les 5 types d'itérateurs, seuls deux sont utilisés pour les conteneurs : les *bidirectional iterators* et les *random access iterators*. Voyons ce qu'ils nous proposent.

Les bidirectional iterators

Ce sont les plus simples des deux. "Bidirectional iterator" signifie itérateur bidirectionnel, mais cela ne nous avance pas beaucoup... 🙄

Ce sont des itérateurs qui permettent d'avancer et de reculer sur le conteneur. Cela veut dire que vous pouvez utiliser aussi bien `++` que `--`. L'important étant que l'on ne peut avancer que d'un seul élément à la fois. Donc pour accéder au 6^{ème} d'un conteneur, il faut partir de la position `begin()` puis appeler cinq fois l'opérateur `++`.

Ce sont les itérateurs utilisés pour les `list`, `set` et `map`. On ne peut donc pas utiliser ces itérateurs pour accéder directement au milieu d'un de ces conteneurs.

Les random access iterators

Au vu du nom, vous vous en doutez peut-être, ces itérateurs permettent d'accéder au hasard, ce qui en meilleur français veut dire que l'on peut accéder directement au milieu d'un conteneur.

Techniquement, ces itérateurs proposent en plus de ++ et -- des opérateurs + et - permettant d'avancer d'un coup de plusieurs éléments.

Par exemple pour accéder au 8^{ème} élément d'un `vector`, on peut utiliser la syntaxe suivante :

Code : C++

```
vector<int> tab(100,2); //Un tableau de 100 entiers valant 2

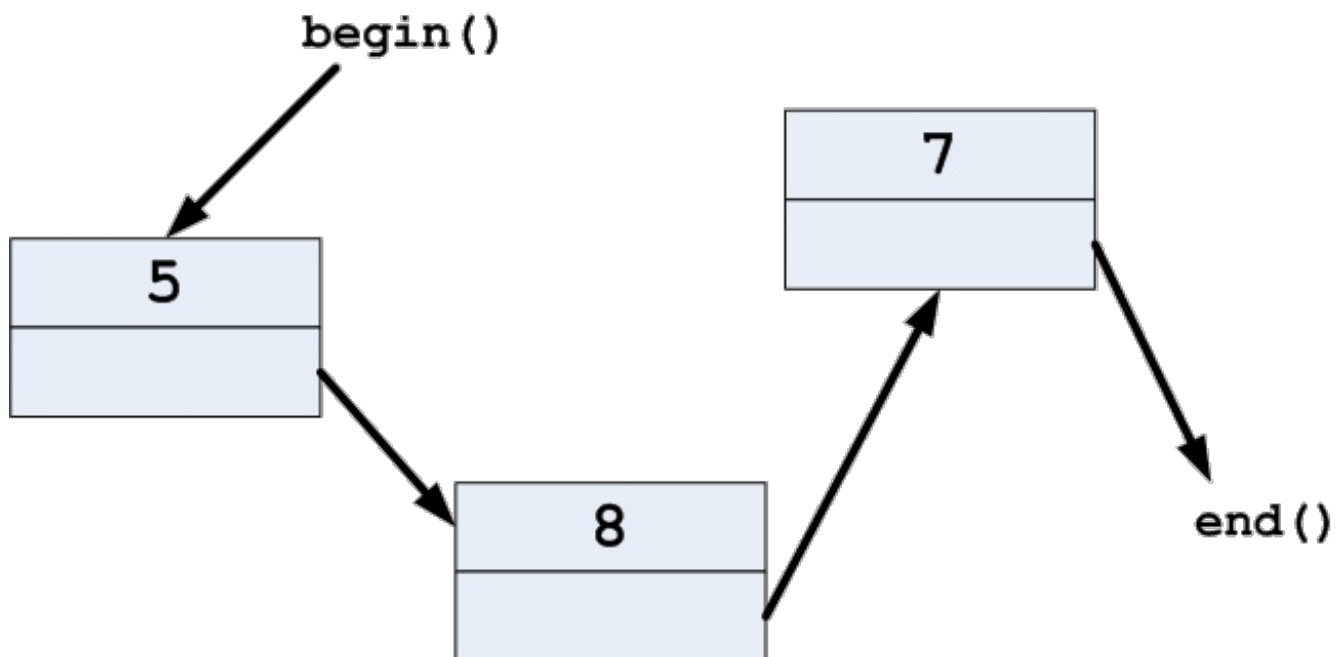
vector<int>::iterator it = tab.begin() + 7; //Un itérateur sur le
8ème élément
```

En plus des `vector`, ces itérateurs sont ceux utilisés par les `deque`.

Le mécanisme exact des itérateurs est très compliqué, c'est pour cela que je ne vous présente que les éléments qui vous seront réellement nécessaires dans la suite. Savoir que certains itérateurs sont plus limités que d'autres nous sera utile au chapitre suivant puisque certains algorithmes ne sont utilisables qu'avec des random access iterators.

La pleine puissance des `list` et `map`

Je ne vous ai pas encore parlé des listes chaînées de type `list`. C'est un conteneur assez différent de ce que vous connaissez. Les éléments ne sont pas rangés les uns à côté des autres dans la mémoire. Chaque "case" contient un élément et un pointeur sur la prochaine case située ailleurs dans la mémoire, comme sur l'illustration suivante :



L'avantage de cette structure de données est que l'on peut facilement ajouter des éléments au milieu. Il n'est pas nécessaire de décaler toute la suite comme dans l'exemple de la bibliothèque du chapitre précédent. Mais, (il y a toujours un mais) on ne peut pas directement accéder à une case donnée... tout simplement parce qu'on ne sait pas où elle se trouve dans la mémoire. 😊 On

est obligé de suivre toute la chaîne des éléments. Pour aller à la 8^{ème} case, il faut aller à la première case, suivre le pointeur jusqu'à la deuxième, suivre le pointeur jusqu'à la troisième et ainsi de suite jusqu'à la 8^{ème}. C'est donc très coûteux.

Passer de case en case dans l'ordre est une mission parfaite pour les itérateurs. 😎 Et puis, il n'y a pas d'opérateur [] pour les listes. On n'a donc pas le choix !

L'avantage c'est que tout se passe comme pour les autres conteneurs. C'est ça la magie des itérateurs. On n'a pas besoin de connaître les spécificités du conteneur pour itérer dessus.

Code : C++ - Manipuler une liste

```

#include <list>
#include <iostream>
using namespace std;

int main()
{
    list<int> liste;           //Une liste d'entiers
    liste.push_back(5);       //On ajoute un entier dans la liste
    liste.push_back(8);       //Et un deuxième
    liste.push_back(7);       //Et encore un !

    //On itère sur la liste
    for(list<int>::iterator it = liste.begin(); it!=liste.end();
    ++it)
    {
        cout << *it << endl;
    }
    return 0;
}

```

Super non ?

La même chose pour les map

La structure interne des map est encore plus compliquée que celle des list. Elles utilisent ce qu'on appelle des arbres binaires et se déplacer dans cet arbre peut vite devenir un vrai casse-tête. Grâce aux itérateurs, ce n'est pas à vous de vous préoccuper de tout ça. Vous utilisez simplement les opérateurs ++ et -- et l'itérateur saute d'élément en élément. Toutes les opérations complexes sont masquées à l'utilisateur.

Il y a juste une petite subtilité avec les tables associatives. Chaque élément est en réalité constitué d'une clé et d'une valeur. Un itérateur ne peut pointer que sur une seule chose à la fois. Il y a donc, a priori, un problème. 😬 Rien de grave je vous rassure.

Les itérateurs pointent en réalité sur des pair. Ce sont des objets avec deux attributs publics appelés first et second. Les pair sont déclarées dans le fichier d'en-tête utility. Il est cependant très rare de devoir utiliser directement ce fichier puisqu'il est inclus par presque tous les autres. 😊 Créons quand même une paire juste pour essayer.

Code : C++ - Utilisation d'une paire

```

#include <utility>
#include <iostream>
using namespace std;

int main()
{
    pair<int, double> p(2, 3.14);    //Une paire contenant un entier
    //valant 2 et un nombre à virgule valant 3.14

    cout << "La paire vaut (" << p.first << ", " << p.second << ")"
    << endl;

    return 0;
}

```

Et c'est tout ! 😬 On ne peut rien faire d'autre avec une paire. Elles servent juste à contenir deux objets.



Les deux attributs sont publics. Cela peut vous sembler bizarre puisque je vous ai conseillé de toujours déclarer vos attributs dans la partie privée de la classe. Les pair sont là juste pour contenir deux variables d'un coup. Il n'y a aucune méthode ni rien dans la classe. C'est juste un outil très basique et on n'a pas envie de s'embêter avec des

méthodes `get()` et `set()`. C'est pour cela que les attributs sont publics.

Dans une map, les objets stockés sont en réalité des *pair*. Pour chaque paire, l'attribut `first` correspond à la clé alors que `second` est la valeur.

Je vous ai dit dans le chapitre précédent que les map triaient leurs éléments selon leurs clés. Nous allons maintenant pouvoir le vérifier facilement.

Code : C++ - Itération sur une map

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    map<string, double> poids; //Une table qui associe le nom d'un
    animal à son poids

    //On ajoute les poids de quelques animaux
    poids["souris"] = 0.05;
    poids["tigre"] = 200;
    poids["chat"] = 3;
    poids["elephant"] = 10000;

    //Et on parcourt la table en affichant le nom et le poids
    for(map<string, double>::iterator it=poids.begin();
    it!=poids.end(); ++it)
    {
        cout << it->first << " pese " << it->second << " kg." <<
    endl;
    }
    return 0;
}
```

Si vous testez, vous verrez que les animaux sont affichés par ordre alphabétique même si on les a insérés dans un tout autre ordre :

Code : Console

```
chat pese 3 kg.
elephant pese 10000 kg.
souris pese 0.05 kg.
tigre pese 200 kg.
```



La map utilise l'opérateur `<` de la classe `string` pour trier ses éléments. Nous verrons dans la suite comment changer ce comportement.

Les itérateurs sont aussi utiles pour rechercher quelque chose dans une table associative. Utiliser l'opérateur `[]` permet d'accéder à un élément donné. Mais il a un "défaut". Si l'élément n'existe pas, l'opérateur `[]` le crée. On ne peut pas l'utiliser pour savoir si un élément donné est déjà présent dans la table ou pas.

C'est pour palier ce problème que les map proposent une méthode `find()` qui renvoie un itérateur sur l'élément recherché. Si l'élément n'existe pas, elle renvoie simplement `end()`. Vérifier si une clé existe déjà dans une table est donc très simple.

Reprenons la table de l'exemple précédent et vérifions si le poids d'un chien s'y trouve.

Code : C++ - Recherche dans une map

```

int main()
{
    map<string, double> poids; //Une table qui associe le nom d'un
    animal à son poids

    //On ajoute les poids de quelques animaux
    poids["souris"] = 0.05;
    poids["tigre"] = 200;
    poids["chat"] = 3;
    poids["elephant"] = 10000;

    map<string, double>::iterator trouve = poids.find("chien");

    if(trouve == poids.end())
    {
        cout << "Le poids du chien n'est pas dans la table" << endl;
    }
    else
    {
        cout << "Le chien pese " << trouve->second << " kg." <<
endl;
    }
    return 0;
}

```

Je crois ne pas avoir besoin d'en dire plus. 😊 Je sens que vous êtes déjà des fans des itérateurs.

Foncteur : la version objet des fonctions

Si vous suivez un cours d'informatique à l'université, on vous dira que les itérateurs sont des abstractions des pointeurs et que les foncteurs sont des abstractions des fonctions. Et généralement, le cours va s'arrêter là. 😊

Je pourrais faire de même et vous laisser vous débrouiller avec un ou deux exemples, mais je ne pense pas que vous seriez très heureux.

Ce que l'on aimerait faire, c'est appliquer des changements sur des conteneurs. Par exemple prendre un tableau de lettres et les convertir toutes en majuscule. Ou prendre une liste de nombres et ajouter 5 à tous les nombres pairs. Bref, on aimerait appliquer une fonction sur tous les éléments d'un conteneur. Le problème c'est qu'il faudrait pouvoir passer cette fonction en argument d'une méthode du conteneur. Et ça, on ne sait pas le faire. On ne peut passer que des objets en argument et pas des fonctions.



Techniquement, ce n'est pas vrai. Il existe des pointeurs sur des fonctions et l'on pourrait utiliser ces pointeurs pour résoudre ce problème. Les foncteurs sont par contre plus simples d'utilisation et offrent plus de possibilités.

Les foncteurs sont des objets possédant une surcharge de l'opérateur (). Ils peuvent ainsi agir comme une fonction mais peuvent être passés en argument à une méthode ou à une autre fonction.

Créer un foncteur

Un foncteur est une classe possédant si nécessaires des attributs et des méthodes. Mais en plus de ça, elle doit proposer un opérateur () qui effectue l'opération que l'on souhaite.

Commençons avec un exemple simple, un foncteur qui additionne deux entiers.

Code : C++ - Un premier foncteur

```

class Addition{
public:

    int operator () (int a, int b)    //La surcharge de l'opérateur ()
    {
        return a+b;
    }
};

```

Cette classe ne possède pas d'attributs et juste une seule méthode, la fameuse surcharge de l'opérateur (). Comme il n'y a pas d'attributs et rien de spécial à effectuer, le constructeur généré par le compilateur est largement suffisant.



Vous aurez reconnu la syntaxe habituelle pour les opérateurs : le mot *operator* suivi de l'opérateur que l'on veut, ici les parenthèses. La particularité de cet opérateur est qu'il peut prendre autant d'arguments que l'on veut au contraire de tous les autres qui ont un nombre d'arguments fixé.

On peut alors utiliser ce foncteur pour additionner deux nombres :

Code : C++ - Utilisation du foncteur

```
#include <iostream>
using namespace std;

int main()
{
    Addition foncteur;
    int a(2), b(3);
    cout << a << " + " << b << " = " << foncteur(a,b) << endl; //On
    utilise le foncteur comme si il s'agissait d'une fonction
    return 0;
}
```

Ce code donne bien évidemment le résultat escompté :

Code : Console

```
2 + 3 = 5
```

Et l'on peut bien sûr créer tout ce que l'on veut comme foncteur. Par exemple, un foncteur ajoutant 5 aux nombres pairs peut être écrit comme suit :

Code : C++ - Un foncteur qui ajoute 5 aux nombres pairs

```
class Ajout{
public:

    int operator () (int a)    //La surcharge de l'opérateur ()
    {
        if(a%2 == 0)
            return a+5;
        else
            return a;
    }
};
```

Rien de neuf en somme ! 😊

Des foncteurs évolutifs

Les foncteurs sont des objets. Ils peuvent donc utiliser des attributs comme n'importe quelle autre classe. Cela nous permet en quelque sorte de créer des fonctions avec une mémoire. Elle pourra donc effectuer une opération différente à chaque appel. Je pense qu'un exemple sera plus parlant.

Code : C++

```
class Remplir{
public:
    Remplir(int i)
        :m_valeur(i)
    {}

    int operator() ()
    {
        ++m_valeur;
        return m_valeur;
    }

private:
    int m_valeur;
};
```

La première chose à remarquer est que notre foncteur possède un constructeur. Son but est simplement d'initialiser correctement l'attribut `m_valeur`. L'opérateur parenthèse renvoie simplement la valeur de cet attribut, mais ce n'est pas tout. Il incrémente cet attribut à chaque appel. Notre foncteur renvoie donc une valeur différente à chaque appel !

On peut par exemple l'utiliser pour remplir un `vector` avec les nombres de 1 à 100. Je vous laisse essayer. 😊

Bon, comme c'est encore une notion récente pour vous, je vous propose quand même une solution :

Code : C++ - Utiliser Remplir pour remplir un tableau

```
int main()
{
    vector<int> tab(100,0); //Un tableau de 100 cases valant toutes
    0

    Remplir f(0);

    for(vector<int>::iterator it=tab.begin(); it!=tab.end(); ++it)
    {
        *it = f(); //On appelle simplement le foncteur sur chacun
        des éléments du tableau
    }

    return 0;
}
```

Ceci n'est bien sûr qu'un exemple tout simple. On peut créer des foncteurs avec beaucoup d'attributs et des comportements bien plus complexes. On peut aussi ajouter d'autres méthodes pour ré-initialiser `m_valeur` par exemple. Comme ce sont des objets, tout ce que vous savez à leur sujet reste valable !



Si vous connaissez le C, vous aurez peut-être pensé au mot-clé **static** qui autorise le même genre de choses pour les fonctions normales. Le foncteur avec des attributs est la manière de réaliser cela en C++.

Les prédicats

Je sens que vous êtes un peu effrayé par ce nouveau nom barbare. C'est vrai que ce chapitre présente plein de nouvelles choses et qu'il faut un peu de temps pour tout assimiler. Rien de bien compliqué ici, je vous rassure. 😊

Les prédicats sont des foncteurs un peu particuliers. Ce sont des foncteurs prenant *un seul argument* et renvoyant un *booléen*.

Ils servent à tester une propriété particulière de l'objet passé en argument. On les utilise pour répondre à des questions comme :

- Ce nombre est-il plus grand que 10 ?
- Cette chaîne de caractère contient-elle des voyelles ?
- Ce Personnage est-il encore vivant ?

Ces prédicats seront très utiles dans la suite. Nous verrons dans le chapitre suivant comment supprimer des objets qui vérifient une certaine propriété. Et c'est bien sûr un foncteur de ce genre qu'il faudra utiliser !

Voyons quand même un petit code avant d'aller plus loin. Prenons le cas d'un prédicat qui teste si une chaîne de caractère contient des voyelles.

Code : C++ - Mon premier prédicat

```
class TestVoyelles
{
public:
    bool operator()(string const& chaine) const
    {
        for(int i(0); i<chaine.size(); ++i)
        {
            switch (chaine[i])    //On teste les lettres une-à-une
            {
                case 'a':        //Si c'est une voyelle
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                case 'y':
                    return true; //On renvoie 'true'
                default:
                    break;        //Sinon, on continue
            }
        }
        return false; //Si on arrive là, c'est qu'il n'y avait pas
// de voyelles du tout
    }
};
```



Nous verrons dans la suite comment écrire cela de manière plus simple !

Terminons cette sous-partie en jetant un œil à quelques foncteurs pré-définis dans la STL. Eh oui, il y en a même pour les fainéants ! 😊

Les foncteurs pré-définis

Pour les opérations les plus simples, le travail est pré-mâché. Tout se trouve dans le fichier d'en-tête `functional`. Je ne vais pas vous présenter tout ce qui s'y trouve dans ce tuto. Je vous propose de faire un tour dans la documentation, cela vous apprendra à vous débrouiller avec ces sites webs.

Prenons tout de même un exemple. Le premier foncteur que je vous ai présenté prenait deux entiers en argument et renvoyait la somme des nombres. La STL propose un foncteur nommé `plus` (quelle originalité 😊) pour faire ça.

Code : C++

```
#include <iostream>
#include <functional> //Ne pas oublier !
using namespace std;

int main()
```

```
{
    plus<int> foncteur;    //On déclare le foncteur additionnant
    deux entiers
    int a(2), b(3);
    cout << a << " + " << b << " = " << foncteur(a,b) << endl; //On
    utilise le foncteur comme si il s'agissait d'une fonction
    return 0;
}
```

Comme pour les conteneurs, il faut indiquer le type souhaité entre les chevrons. En utilisant ces foncteurs pré-définis, on s'économise un peu de travail. 😊

Voyons finalement comment utiliser ces foncteurs avec des conteneurs.

Fusion des deux concepts

Les foncteurs sont au cœur de la STL. Ils sont très utilisés dans les algorithmes que nous verrons dans le prochain chapitre. Pour l'instant, nous allons modifier le critère de tri des `map` grâce à un foncteur.

Modifier le comportement d'une map

Le constructeur de la classe `map` prend en réalité un argument : le foncteur de comparaison entre les clés. Par défaut, si l'on ne spécifie rien, c'est un foncteur construit à partir de l'opérateur `<` qui sert de comparaison. La `map` que nous avons utilisée précédemment utilisait ce foncteur par défaut.

L'opérateur `<` pour les `string` compare les chaînes par ordre alphabétique. Changeons ce comportement pour utiliser une comparaison de longueur. Je vous laisse essayer d'écrire un foncteur comparant la longueur de deux `string`.

Voici ma solution :

Code : C++

```
#include <string>
using namespace std;

class CompareLongueur
{
public:
    bool operator()(const string& a, const string& b)
    {
        return a.length() < b.length();
    }
};
```

Je pense que vous avez écrit quelque chose de similaire. 😊

Il ne reste maintenant plus qu'à indiquer à notre `map` que nous voulons utiliser ce foncteur.

Code : C++

```
int main()
{
    //Une table qui associe le nom d'un animal à son poids
    map<string, double, CompareLongueur> poids; //On utilise le
    foncteur comme critère de comparaison

    //On ajoute les poids de quelques animaux
    poids["souris"] = 0.05;
    poids["tigre"] = 200;
    poids["chat"] = 3;
    poids["elephant"] = 10000;
```

```
//Et on parcourt la table en affichant le nom et le poids
for (map<string, double>::iterator it=poids.begin();
it!=poids.end(); ++it)
{
    cout << it->first << " pese " << it->second << " kg." << endl;
}
return 0;
}
```

Et ce programme donne le résultat suivant :

Code : Console

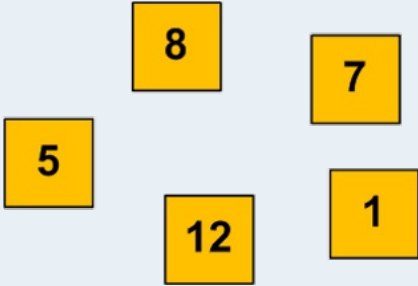
```
chat pese 3 kg.
tigre pese 200 kg.
souris pese 0.05 kg.
elephant pese 10000 kg.
```

Les animaux ont été triés selon la longueur de leur nom. 😊 Changer le comportement d'un conteneur est donc une opération très simple à réaliser.

Récapitulatif des conteneurs les plus courants

Dans le chapitre suivant, nous allons utiliser plusieurs conteneurs différents et comme tout ça est encore un peu nouveau pour vous, voici un petit tableau récapitulatif des 5 conteneurs les plus courants.

Conteneur	Caractéristiques	Exemple	Schéma
vector	<ul style="list-style-type: none"> Éléments stockés côte-à-côte. Optimisé pour l'ajout en fin de tableau. Éléments indexés par des entiers. 	<code>vector<int></code>	
deque	<ul style="list-style-type: none"> Éléments stockés côte-à-côte. Optimisé pour l'ajout en début et en fin de tableau. Éléments indexés par des entiers. 	<code>deque<int></code>	
list	<ul style="list-style-type: none"> Éléments stockés de manière "aléatoire" dans la mémoire. Ne se parcourt qu'avec des itérateurs. Optimisé pour l'insertion et la suppression au milieu. 	<code>list<int></code>	
map	<ul style="list-style-type: none"> Éléments indexés par ce que l'on veut. Éléments triés selon leurs index. Ne se parcourt qu'avec des itérateurs. 	<code>map<string, int></code>	

set	<ul style="list-style-type: none">• Éléments triés.• Ne se parcourt qu'avec des itérateurs.	<code>set<int></code>	
------------	--------------------------------------------------------------------------------------------------------------------	-----------------------------	-------------------------------------------------------------------------------------

Ce chapitre ne présente que la base des itérateurs. Il y aurait encore beaucoup de choses à dire sur le sujet. 😎 Mais pour l'utilisation que nous allons en faire dans la suite, vous savez tout.

Dans le chapitre suivant, nous allons découvrir une série d'opérations que l'on peut effectuer sur les éléments d'un conteneur. Et bien sûr les itérateurs et foncteurs vont apparaître.

La puissance des algorithmes

Nous avons découvert les itérateurs qui nous permettent de parcourir des conteneurs, comme les `vector`. Dans ce chapitre, nous allons découvrir les *algorithmes* de la STL, des fonctions qui nous permettent d'effectuer des modifications sur les conteneurs.

Cela fait un moment que je vous parle de modifications, mais qu'est-ce que cela veut dire ? Eh bien, par exemple on peut trier un tableau, supprimer les doublons, inverser une sélection, chercher, remplacer ou supprimer des éléments, etc.

Certains de ces algorithmes sont simples à écrire et vous ne voyez peut-être pas l'intérêt d'utiliser des fonctions déjà faites. Après tout, vous savez déjà chercher vous-mêmes quelque chose dans un `vector` par exemple.

L'avantage d'utiliser les algorithmes de la STL est qu'il n'y a pas besoin de réfléchir pour écrire ces fonctions. Il n'y a qu'à utiliser ce qui existe déjà. De plus, ces fonctions sont extrêmement optimisées. En bref, ne réinventez pas la roue et utilisez les algorithmes de la STL que je vais vous présenter ! 😊



Il est nécessaire d'avoir bien compris le chapitre précédent, notamment les itérateurs et les foncteurs. Nous allons en utiliser beaucoup dans ce qui suit. 🤖

Un premier exemple

Je vous prévien tout de suite, nous n'allons pas étudier tous les algorithmes proposés par la STL. Il y en a une soixantaine et ils ne sont pas tous très utiles. Et puis, quand vous aurez compris le principe, vous saurez vous débrouiller seuls. 😊

La première chose à faire est comme toujours l'inclusion du bon en-tête. Dans notre cas, il s'agit du fichier `algorithm`. Et croyez-moi, vous allez souvent en avoir besoin à partir de maintenant.

Un début en douceur

Dans le [chapitre précédent](#), nous avons créé un foncteur nommé `Remplir` et nous l'avons appliqué à tous les éléments d'un `vector`. Cela se faisait via une boucle **for** qui parcourait les éléments du tableau de la position `begin()` à la position `end()`.

Le plus simple des algorithmes s'appelle `generate` et fait exactement la même chose, mais de façon plus optimisée. Il appelle un foncteur sur tous les éléments situés entre deux itérateurs. Grâce à cet algorithme, notre code de remplissage de tableau devient beaucoup plus court :

Code : C++

```
#include <algorithm>
#include <vector>
using namespace std;

//Définition de Remplir...

int main()
{
    vector<int> tab(100,0); //Un tableau de 100 cases valant toutes
    0

    Remplir f(0);

    generate(tab.begin(), tab.end(), f); //On applique f à tout ce qui
    se trouve entre begin() et end()

    return 0;
}
```

Ce code a l'avantage d'être en plus très simple à comprendre. 😊 Si vous parlez la langue de Shakespeare, vous aurez compris que "to generate" signifie "générer". La ligne mise en évidence se lit donc de la manière suivante : *Génère grâce au foncteur f tous les éléments situés entre tab.begin() et tab.end()*. On peut difficilement faire plus clair !



Mais pourquoi doit-on utiliser des itérateurs ? Pourquoi la fonction `generate()` ne prend-elle pas comme premier argument le `vector` ?

Excellente question ! 😊 Je vois que vous suivez. Il serait bien plus simple de pouvoir écrire quelque chose comme `generate(tab, f)` à la place des itérateurs. On s'éviterait toute la théorie sur les itérateurs !

En fait c'est une fausse bonne idée que de faire comme ça. Imaginez que vous ne vouliez appliquer votre foncteur qu'aux dix premiers éléments du tableau et pas au tableau entier. Comment feriez-vous avec votre technique ? 😊 Ce ne serait tout simplement pas possible. L'avantage des itérateurs est clair dans ce cas : on peut se restreindre à une portion d'un conteneur. Tenez, pour remplir seulement les 10 premières cases, on ferait ceci :

Code : C++

```
int main()
{
    vector<int> tab(100,0); //Un tableau de 100 cases valant toutes
    0

    Remplir f(0);

    generate(tab.begin(), tab.begin()+10, f); //On applique f aux
10 premières cases
    generate(tab.end()-5, tab.end(), f);      //Et aux 5 dernières

    return 0;
}
```

Plutôt sympa non ?

En fait, c'est une propriété importante des algorithmes, ils s'utilisent toujours sur une plage d'éléments situés entre deux itérateurs.

Application aux autres conteneurs

Le deuxième avantage d'utiliser des itérateurs est qu'ils existent pour tous les conteneurs. On peut donc utiliser les algorithmes sur tous les types de conteneurs ou presque. Il existe quand même quelques restrictions selon que les itérateurs sont aléatoires bidirectionnels ou constants (pour les `set` notamment) comme on l'a vu dans le chapitre précédent.

Par exemple, on peut tout à fait utiliser notre foncteur sur un `list<int>`.

Code : C++

```
int main()
{
    list<int> tab; //Un ensemble d'entiers

    //Quelques manipulations pour créer des éléments...

    Remplir f(0);

    generate(tab.begin(), tab.end(), f); //On applique f aux
éléments de l'ensemble

    return 0;
}
```

La syntaxe est strictement identique ! Il suffit donc de comprendre une fois le fonctionnement de tout ceci pour pouvoir effectuer des manipulations complexes sur n'importe quel type de conteneur ! 🧙



Il faut quand même que le foncteur corresponde au type contenu. On ne peut bien sûr pas utiliser un foncteur manipulant des `string` sur une deque de nombres à virgule. Il faut rester raisonnable. Le compilateur génère parfois des message d'erreur très difficiles à interpréter quand on se trompe avec la STL. Soyez donc vigilants.

Compter, chercher, trier

Bon, plongeons-nous un peu plus en avant dans la documentation de l'en-tête `algorithm`. Commençons par quelques fonctions de comptage.

Compter des éléments

Compter des éléments est une opération très facile à réaliser. Utiliser la STL peut à nouveau vous sembler superflu, moi je trouve que cela rend le code plus clair et peut-être même plus optimisé dans certains cas.

Pour compter le nombre d'éléments égaux à une valeur donnée, on utilise l'algorithme `count`. Oui, être anglophone aide beaucoup en programmation. Mais je crois que vous l'avez compris. 🤔

Pour compter le nombre d'éléments égaux au nombre 2, c'est très simple :

Code : C++

```
int nombre = count(tab.begin(), tab.end(), 2);
```

Et bien sûr `tab` est le conteneur de votre choix. Et voilà, vous savez tout ! 🤖 En tout cas pour cet algorithme...

Avant d'aller plus loin, faisons un petit exercice pour récapituler tout ce que nous savons sur les foncteurs, `generate()` et `count()`. Essayez d'écrire un programme qui génère un tableau de 100 nombres aléatoires entre 0 et 9 puis qui compte le nombre de 5 générés. Tout ceci en utilisant au maximum la STL bien sûr ! A vos claviers !

Vous avez réussi ? Voici une solution possible :

Code : C++

```
#include <iostream>
#include <cstdlib> //pour rand()
#include <ctime> //pour time()
#include <vector>
#include <algorithm>
using namespace std;

class Generer
{
public:
    int operator() () const
    {
        return rand() % 10; //On renvoie un nombre entre 0 et 9
    }
};

int main()
{
    srand(time(0));

    vector<int> tab(100,-1); //Un tableau de 100 cases

    generate(tab.begin(), tab.end(), Generer()); //On
    génère les nombres aléatoires

    int const compteur = count(tab.begin(), tab.end(), 5); //Et on
    compte les occurrences du 5

    cout << "Il y a " << compteur << " elements valant 5." << endl;
```



```

    return 0;
}

```

Personnellement, je trouve ce code très clair. On voit rapidement ce qui se passe. Toutes les boucles nécessaires sont cachées dans les fonctions de la STL. Pas besoin de s'ennuyer à devoir tout écrire soi-même.

Le retour des prédicats

Si vous pensiez que vous pourriez vous en sortir sans ces drôles de foncteurs, vous vous trompiez ! 😊 Je vous avais dit dans le chapitre précédent que l'on utilisait des prédicats pour tester une propriété des éléments. On pourrait donc utiliser un prédicat pour ne compter que les éléments qui passent un certain test. Et si je vous en parle, c'est qu'un tel algorithme existe. Il s'appelle `count_if()`. La différence avec `count()` est que le troisième argument n'est pas une valeur mais un prédicat.

Dans le chapitre précédent, nous avons écrit un prédicat qui testait si une chaîne de caractères contenait des voyelles ou non. Essayons-le !

Code : C++

```

#include <algorithm>
#include <string>
#include <vector>
using namespace std;

class TestVoyelles
{
public:
    bool operator()(string const& chaine) const
    {
        for(int i(0); i<chaine.size(); ++i)
        {
            switch (chaine[i])    //On teste les lettres une-à-une
            {
                case 'a':        //Si c'est une voyelle
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                case 'y':
                    return true; //On renvoie 'true'
                default:
                    break;       //Sinon, on continue
            }
        }
        return false; //Si on arrive là, c'est qu'il n'y avait pas
de voyelles du tout
    }
};

int main()
{
    vector<string> tableau;

    //... On remplit le tableau en lisant un fichier par exemple.

    int const compteur = count_if(tableau.begin(), tableau.end(),
TestVoyelles());

    //... Et on fait quelque chose avec 'compteur'

    return 0;
}

```

Voilà qui est vraiment puissant ! Le prédicat `TestVoyelles` s'active sur chacun des éléments du tableau et `count_if` indique combien de fois le prédicat a renvoyé "vrai". On sait ainsi combien il y a de chaînes contenant des voyelles dans le tableau. 😊

Chercher

Chercher un élément dans un tableau est aussi très facile. On utilise l'algorithme `find()` ou `find_if()`. Ils s'utilisent exactement comme les algorithmes de comptage, la seule différence est leur type de retour : ils renvoient un itérateur sur l'élément trouvé ou sur `end()` si l'objet cherché n'a pas été trouvé.

Pour chercher la lettre **a** dans une deque de `char`, on fera quelque chose comme :

Code : C++

```
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    deque<char> lettres;

    //On remplit la deque... avec generate() par exemple !

    deque<char>::iterator trouve = find(lettres.begin(),
    lettres.end(), 'a');

    if(trouve == lettres.end())
        cout << "La lettre 'a' n'a pas ete trouvee" << endl;
    else
        cout << "La lettre 'a' a ete trouvee" << endl;

    return 0;
}
```

Et je ne vous fais pas l'affront de vous montrer la version qui utilise un prédicat. 😊 Je suis convaincu que vous saurez vous débrouiller.

Puisque l'on parle de recherche d'éléments, je vous signale juste l'existence des fonctions `min_element()` et `max_element()` qui cherchent l'élément le plus petit ou le plus grand.

Trier !

Il arrive souvent que l'on doive trier une série d'éléments. Et ce n'est pas une mince affaire. En tout cas, c'est un problème avancé d'algorithmique (la science des algorithmes 😊). Je vous assure qu'écrire une fonction de tri optimisée est une tâche qui n'est pas à la portée de beaucoup de monde. 😊

Heureusement, la STL propose une fonction pour cela, et je peux vous assurer qu'elle est très efficace et bien codée. Son nom est simplement `sort()`, ce qui signifie *trier* en anglais (au cas où je devrais le préciser).

On lui fournit deux itérateurs et la fonction va trier tout ce qui se trouve entre ces deux éléments dans l'ordre croissant. Trions donc le tableau de nombres aléatoires utilisé précédemment.

Code : C++

```
int main()
{
```

```

    srand(time(0));

    vector<int> tab(100,-1); //Un tableau de 100 cases

    generate(tab.begin(), tab.end(), Generer());           //On
    génère les nombres aléatoires

    sort(tab.begin(), tab.end());                           //On
    trie le tableau

    for(vector<int>::iterator it=tab.begin(); it!=tab.end(); ++it)
        cout << *it << endl;
    //On affiche le tableau trié

    return 0;
}

```

A nouveau, rien de bien sorcier. 😊



La fonction `sort()` ne peut être utilisée qu'avec des conteneurs proposant des *random access iterators*, c'est-à-dire les `vector` et les `deque` uniquement. De toute façon, trier une `map` a peu de sens puisque ces conteneurs stockent directement leurs éléments dans le bon ordre. 😊

Par défaut, la fonction `sort()` utilise l'opérateur `<` pour comparer les éléments avant de les trier. Mais il existe également une autre version de cette fonction qui prend un troisième argument : un foncteur comparant deux éléments. Nous avons déjà rencontré un tel foncteur dans le chapitre précédent pour changer le comportement d'une table associative. C'est exactement le même principe ici. Si l'on souhaite créer un tri spécifique, on doit fournir un foncteur expliquant à `sort()` comment trier.

Pour trier des chaînes de caractères selon leur longueur, nous pouvons ré-utiliser notre foncteur :

Code : C++

```

class ComparaisonLongueur
{
public:
    bool operator()(const string& a, const string& b)
    {
        return a.length() < b.length();
    }
};

int main()
{
    vector<string> tableau;

    //... On remplit le tableau en lisant un fichier par exemple.

    sort(tableau.begin(), tableau.end(), ComparaisonLongueur());

    //Le tableau est maintenant trié par longueur de chaîne

    return 0;
}

```

Puissant, simple et efficace. Que demander de mieux ?

Encore plus d'algos

Ne nous arrêtons pas en si bon chemin. On est encore loin d'avoir fait le tour de tout ce qui existe.

Dans l'exemple du tri, j'affichais le contenu du `vector` via une boucle `for`. Faire cela via un algorithme serait plus élégant. 😊

Concrètement, afficher les éléments revient à les passer en argument à une fonction (ou un foncteur) qui les affiche. Écrire un foncteur qui affiche l'argument reçu ne devrait pas vous poser de problèmes à ce stade du cours.

Code : C++

```
class Afficher
{
public:
    void operator() (int a) const
    {
        cout << a << endl;
    }
};
```

Il ne nous reste plus qu'à appliquer ce foncteur sur tous les éléments. L'algorithme pour faire ça s'appelle `for_each()`, ce qui signifie "pour tout".

Code : C++

```
int main()
{
    srand(time(0));
    vector<int> tab(100, -1);
    generate(tab.begin(), tab.end(), Generer()); //On génère des
    nombres aléatoires
    sort(tab.begin(), tab.end());

    for_each(tab.begin(), tab.end(), Afficher()); //Et on affiche
    les éléments

    return 0;
}
```

On a encore raccourci notre code. 😊

A partir de cet algorithme, on peut faire énormément de choses. Un des premiers cas qui me vient à l'esprit est le calcul de la somme des éléments d'un conteneur. Vous voyez comment ? Comme `for_each()` appelle le foncteur sur tous les éléments de la plage spécifiée, on peut demander au foncteur de sommer les éléments dans un de ses attributs.

Code : C++

```
class Sommer
{
public:
    Sommer()
        :m_somme(0)
    {}

    void operator() (int n)
    {
        m_somme += n;
    }

    int resultat() const
    {
        return m_somme;
    }

private:
    int m_somme;
```

```
};
```

L'opérateur `()` va simplement ajouter la valeur de l'élément courant à l'attribut `m_somme`. Après l'appel à l'algorithme, on peut consulter la valeur de `m_somme` en utilisant la méthode `resultat()`.

Il faut cependant faire attention. La fonction `for_each` reçoit une copie du foncteur en argument et pas une référence. Cela veut dire que l'objet dont l'attribut `m_somme` est incrémenté n'est pas celui déclaré dans le `main`, mais une copie de celui-ci. 😞

Heureusement pour nous, les concepteurs de la STL ont pensé à tout et la fonction `for_each()` renvoie le foncteur qu'elle a utilisé une fois qu'elle a terminé. On peut donc utiliser l'objet retourné pour connaître la somme.

Code : C++

```
int main()
{
    srand(time(0));
    vector<int> tab(100, -1);
    generate(tab.begin(), tab.end(), Generer()); //On génère des
    nombres aléatoires

    Sommer somme;

    somme = for_each(tab.begin(), tab.end(), somme); //On somme
    les éléments et on récupère le foncteur utilisé

    cout << "La somme des elements generes est : " <<
    somme.resultat() << endl;

    return 0;
}
```

Si vous voulez un exercice, je peux vous proposer de récrire la fonction qui [calculait la moyenne d'un tableau de notes](#) que nous avons vu au tout début de ce cours. Un petit foncteur pour le calcul de la moyenne, un `for_each()` et le tour est joué. 😊

Utiliser deux séries à la fois

Terminons cette courte présentation avec un dernier algorithme bien pratique pour traiter deux conteneurs à la fois. Imaginons que nous voulions calculer la somme des éléments de deux tableaux et stocker le résultat dans un troisième `vector`. Pour cela, il va nous falloir un foncteur qui effectue l'addition. Mais ça, on l'a déjà vu, ça existe dans l'en-tête `functional`. Pour le reste, il nous faut parcourir en parallèle deux tableaux et écrire les résultats dans un troisième. C'est ce que fait la fonction `transform()`. Elle prend 5 arguments. Le début et la fin du premier tableau, le début du deuxième, le début de celui où seront stockés les résultats et bien sûr le foncteur. 🤔

Code : C++

```
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
    vector<double> a(50, 0.); //Trois tableaux de 50 nombres à
    virgule
    vector<double> b(50, 0.);
    vector<double> c(50, 0.);

    //Remplissage des vectors 'a' et 'b'....

    transform(a.begin(), a.end(), b.begin(), c.begin(),
    plus<double>());
```

```
//A partir d'ici les cases de 'c' contiennent la somme des  
cases de 'a' et 'b'  
  
    return 0;  
}
```



Il faut tout de même que les tableaux `b` et `c` soient assez grands. Si ils ont moins de 50 cases (la taille de `a`), ce code va planter lors de l'exécution puisque l'algorithme va tenter de remplir des cases inexistantes.

Arrêtons-nous là pour ce chapitre. Je vous ai parlé des algorithmes les plus utilisés et je pense que vous avez compris comment tout cela fonctionnait. Vous commencez à avoir une bonne expérience du langage. 😊

Je n'ai bien sûr pas pu vous présenter tous les algorithmes. Au travers de quelques exemples vous avez vu comment combiner les itérateurs, les foncteurs et les prédicats pour réaliser des opérations plutôt compliquées... et tout ça en gardant un code source très lisible et facile à comprendre.

Il est rare de se trouver dans un cas où la STL ne propose pas d'algorithme adapté. Je vous propose d'ailleurs d'aller faire un tour dans votre documentation favorite pour consulter la longue liste des fonctions proposées. Pour le site *cplusplus.com*, vous devriez arriver sur cette [page des algorithmes](#).

Avec tout ça, vous allez pouvoir devenir des champions du C++. Savoir utiliser la STL à bon escient est ce qui fait la différence entre un bon programmeur et un excellent programmeur. 😊

Partie 5 : [Théorie] Notions avancées

Nous voici arrivé dans la 5ème partie de ce cours. Nous allons y découvrir quelques notions plus avancées du langage C++. Dans un premier temps, nous allons découvrir un moyen de traiter les erreurs qui peuvent survenir dans un programme puis nous allons parler des templates. C'est un mécanisme assez unique qui nous permet de créer du code utilisable avec différents types. Vous allez enfin pouvoir comprendre le sens des chevrons <et> que l'on voit lorsqu'on définit un `vector<int>` par exemple. 😊

Nous terminerons avec un chapitre présentant quelques points particuliers du langage. Nous parlerons à cette occasion de la fameuse ligne `using namespace std;` que vous connaissez maintenant depuis des lustres.

La gestion des erreurs avec les exceptions

Jusque là, nous avons toujours supposé que tout se déroulait bien dans nos programmes. Mais ce n'est pas toujours le cas, des problèmes peuvent survenir. Pensez par exemple aux cas suivants :

- Un problème à l'ouverture d'un fichier.
- La connexion au serveur de chat qui n'arrive à se faire.
- On a entièrement rempli la mémoire de l'ordinateur.
- On accède à la case n°12 d'un tableau de 7 éléments.

Les exceptions sont un moyen de gérer efficacement les erreurs qui pourraient survenir dans votre programme ; on peut alors tenter de traiter ces erreurs, remettre le programme dans un état normal et reprendre l'exécution du programme.

Dans ce chapitre, je vais vous apprendre à créer des exceptions, à les traiter et à sécuriser vos programmes en les rendant plus robustes.

Un problème bien ennuyeux

En programmation, quel que soit le langage utilisé (et donc en C++ 🤖), il existe plusieurs types d'erreurs qui peuvent survenir. Parmi les erreurs possibles, on connaît déjà les erreurs de syntaxe qui surviennent lorsque l'on fait une faute dans le code source, par exemple si l'on oublie un point-virgule à la fin d'une ligne. Ces erreurs sont facilement corrigées car le compilateur peut les signaler.

Un autre type de problèmes peut survenir si le programme est syntaxiquement correct mais qu'il exécute une action interdite. On peut citer comme exemple les cas où l'on essaye de lire la 10ème case d'un tableau de 8 éléments ou encore le calcul de la racine carrée d'un nombre négatif.

On appelle ces erreurs les **erreurs d'implémentation**.

La gestion des exceptions permet, si elle est réalisée correctement, de corriger les erreurs d'implémentation en les prévoyant à l'avance. Ceci n'est pas toujours réalisable, car il faudrait penser à toutes les erreurs qui pourraient survenir, mais on peut facilement en éviter une grande partie.

Le plus simple pour comprendre le but de la gestion des exceptions est de prendre un exemple concret.

Exemple d'erreur d'implémentation

Cet exemple n'est pas très original (on le trouve dans presque tous les livres), mais c'est certainement parce que c'est un des cas les plus simples.

Imaginons que vous ayez décidé de réaliser une calculatrice. Vous auriez par exemple pu coder la division de deux nombres entiers de cette manière :

Code : C++

```
int division(int a,int b) // Calcule a divisé par b.
{
    return a/b;
}

int main()
{
    int a,b;
    cout << "Valeur pour a : ";
```

```

cin >> a;
cout << "Valeur pour b : ";
cin >> b;

cout << a << " / " << b << " = " << division(a,b) << endl;

return 0;
}

```

Ce code est tout à fait correct et fonctionne parfaitement. Sauf dans un cas : si `b` vaut 0. En effet la division par 0 n'est pas une opération arithmétique valide. Si on lance le programme avec `b=0`, on obtient une erreur et le message suivant s'affiche :

Code : Console

```

Valeur pour a : 3
Valeur pour b : 0
Exception en point flottant (core dumped)

```

Il faudrait donc ne pas réaliser le calcul si `b` vaut 0, mais que faire à la place ?

Quelques mauvaises solutions

Une première possibilité serait de renvoyer un nombre prédéfini à la place du résultat. Ce qui donnerait par exemple :

Code : C++

```

int division(int a,int b) // Calcule a divisé par b.
{
    if( b!=0) // Si b ne vaut pas 0.
        return a/b;
    else // Sinon.
        return ERREUR;
}

```

En spécifiant une valeur précise pour `ERREUR`. Mais cela pose un nouveau problème, quelle valeur choisir pour `ERREUR` ? On ne peut pas renvoyer un nombre puisque tous les nombres pourraient être renvoyés par la fonction dans un cas normal. Ce n'est donc pas une bonne solution.

Une autre idée que l'on rencontre souvent, c'est d'afficher un message d'erreur, ce qui donnerait quelque chose comme :

Code : C++

```

int division(int a,int b) // Calcule a divisé par b.
{
    if( b!=0) // Si b ne vaut pas 0.
        return a/b;
    else // Sinon.
        cout << "ERREUR: Division par 0 !" << endl;
}

```

Mais cela pose deux nouveaux problèmes, la fonction ne renvoie aucune valeur en cas d'erreur et un effet de bord se produit ; en effet la fonction `division` n'est pas forcément censée utiliser `cout` surtout si par exemple, on a réalisé un programme avec une GUI comme Qt par exemple.

La 3^e et dernière solution, que l'on rencontre parfois dans certaines bibliothèques, est de modifier la signature et le type de retour

de la fonction de la manière suivante :

Code : C++

```
bool division(int a, int b, int& resultat)
{
    if (b != 0)           // Si b est différent de 0.
    {
        resultat = a/b;   // On effectue le calcul et on met le
        // résultat dans la variable passée en argument.
        return true;      // On renvoie vrai pour montrer que tout
        // s'est bien passé.
    }
    else                  // Sinon
        return false;     // On renvoie false pour montrer qu'une
        // erreur s'est produite.
}
```

Cette solution est la meilleure des 3 proposées (ceux qui connaissent le C sont habitués à ces choses), mais elle souffre d'un gros problème, elle n'est pas du tout intuitive à utiliser. Il est en particulier impossible de réaliser le calcul $a/(b/c)$ de manière simple et intuitive.



Ces 3 solutions proposées sont là à titre d'illustration de ce qu'il ne faut pas faire. La bonne solution est présentée dans la suite.

La gestion des exceptions

Voyons comment résoudre ce problème de manière élégante en C++.

Principe général

Le principe général des exceptions est le suivant :

- On crée des zones où l'ordinateur va **essayer** le code en sachant qu'une erreur peut survenir.
- Si une erreur survient, on la signale en **lançant** un **objet** qui contient des informations sur l'erreur.
- À l'endroit où l'on souhaite gérer les erreurs survenues, on **attrape** l'objet et on gère l'erreur.

C'est un peu comme si vous étiez coincé sur une île déserte. Vous lanceriez une bouteille à la mer avec des informations dedans permettant de vous retrouver. Il n'y aurait alors plus qu'à espérer que quelqu'un attrape votre bouteille. Sinon vous mourrez de faim.

C'est la même chose ici, on lance un objet en espérant qu'un autre bout de code le rattrapera, sinon le programme plantera.

Dans le principe général, j'ai volontairement mis 3 mots en rouge. Ces 3 mots correspondent aux 3 mots-clés qui sont utilisés par le mécanisme des exceptions.

- **try{ ... }** (essaye en français) permet de signaler une portion de code où une erreur peut survenir.
- **throw** (lance en français) permet de signaler l'erreur en lançant un objet.
- **catch(...){...}** (attrape en français) permet d'introduire la portion de code qui va récupérer l'objet et s'occuper de gérer l'erreur.

Voyons cela plus en détail.

Les 3 mots-clés en détail

Commençons par **try**, il est très simple d'utilisation. Il permet d'introduire un bloc sensible aux exceptions. C'est-à-dire qu'on indique au compilateur qu'une certaine portion du code source pourrait lancer un objet (la bouteille à la mer).

On l'utilise comme ceci :

Code : C++ - Le mot-clé try

```
// Du code sans risque.  
try  
{  
    // Du code qui pourrait créer une erreur.  
}
```

Entre les accolades du bloc try on peut trouver **n'importe quelle instruction C++**, notamment un autre bloc **try**.

Le mot-clé **throw** est lui aussi très simple d'utilisation. C'est grâce à lui qu'on lance notre bouteille. La syntaxe est la suivante :
throw expression

On peut lancer n'importe quoi comme objet, par exemple un **int** qui correspond au numéro de l'erreur ou une string contenant le texte de l'erreur. On verra plus loin un type d'objet particulièrement utile pour les erreurs.

Code : C++ - Le mot-clé throw

```
throw 123; // On lance l'entier 123, par exemple si l'erreur 123  
est survenue.  
  
throw string("Erreur fatale. Contactez un administrateur"); // On  
peut lancer une string.  
  
throw Personnage; // On peut tout à fait lancer une instance d'une  
classe.  
  
throw 3.14 * 5.12; // Ou même le résultat d'un calcul
```



throw peut se trouver n'importe où dans le code, mais s'il n'est pas dans un bloc **try**, l'erreur ne pourra pas être rattrapée et le programme plantera.

Terminons avec le mot-clé **catch**. Il permet de créer un bloc de gestion d'une exception survenue. Il faut créer un bloc **catch** par type d'objet lancé. Chaque bloc **try** doit obligatoirement être suivi d'un bloc **catch**. De manière réciproque, tout bloc **catch** doit être précédé d'un bloc **try** ou d'un autre bloc **catch**.

La syntaxe est la suivante :
catch (type const& e) {}



On attrape les exceptions par référence constante (d'où la présence du **&**) et pas par valeur, ceci afin d'éviter une copie et de conserver le polymorphisme de l'objet reçu. Souvenez-vous des ingrédients nécessaires au polymorphisme, une référence ou un pointeur sont nécessaires. Comme l'objet lancé pourrait avoir des fonctions virtuelles, on l'attrape *via* une référence de sorte que les deux ingrédients soient réunis.

Ce qui donne par exemple :

Code : C++ - Le mot-clé catch

```
try  
{  
    // Le bloc sensible aux erreurs.  
}  
catch(int e) // On rattrape les entiers lancés (pour les entiers,  
une référence n'a pas de sens).  
{  
    // On gère l'erreur.  
}
```

```
catch(string const& e) // On rattrape les strings lancés.
{
    // On gère l'erreur.
}
catch(Personnage const& e) // On rattrape les personnages.
{
    // On gère l'erreur.
}
```



Vous pouvez mettre autant de blocs **catch** que vous voulez. Il en faut **au moins un** par type d'objet pouvant être lancé.



Qu'est-ce que ça va changer durant l'exécution du programme ?

À l'exécution, le programme va se dérouler normalement comme si les instructions **try** et les blocs **catch** n'étaient pas là. Par contre, au moment où l'ordinateur arrive sur une instruction **throw**, il va sauter toutes les instructions suivantes, appeler le destructeur de tous les objets déclarés à l'intérieur du bloc **try**. Il va chercher le bloc **catch** qui correspond à l'objet qui a été lancé.

Arrivé au bloc **catch**, il va exécuter ce qui se trouve dans le bloc et reprendre l'exécution du programme **après** le bloc **catch**.



Je me répète, mais c'est une erreur courante. L'exécution reprend après le bloc catch et pas à l'endroit où se trouve le **throw**.

Le mieux pour comprendre le fonctionnement est encore de reprendre l'exemple de la calculatrice et de la division par 0.

La bonne solution

Reprenons donc notre fonction de calculatrice.

Code : C++

```
int division(int a, int b)
{
    return a/b;
}
```

Nous savons qu'une erreur peut survenir si **b** vaut 0, il faut donc lancer une exception dans ce cas. J'ai choisi, arbitrairement, de lancer une chaîne de caractères. C'est néanmoins un choix intéressant, puisque l'on peut ainsi décrire le problème survenu.

Code : C++

```
int division(int a, int b)
{
    if(b == 0)
        throw string("ERREUR : Division par zéro !");
    else
        return a/b;
}
```

Souvenez-vous, un **throw** doit toujours se trouver dans un bloc **try** qui doit lui-même être suivi d'un bloc **catch**. Ce qui donne la structure suivante :

Code : C++

```

int division(int a,int b)
{
    try
    {
        if(b == 0)
            throw string("Division par zéro !");
        else
            return a/b;
    }
    catch(string const& chaine)
    {
        // On gère l'exception.
    }
}

```

Il ne reste plus alors qu'à gérer l'erreur, c'est-à-dire par exemple, afficher un message d'erreur.

Code : C++

```

int division(int a,int b)
{
    try
    {
        if(b == 0)
            throw string("Division par zéro !");
        else
            return a/b;
    }
    catch(string const& chaine)
    {
        cerr << chaine << endl;
    }
}

```

Ce qui donne le résultat suivant :

Code : Console

```

Valeur pour a : 3
Valeur pour b : 0
ERREUR : Division par zéro !

```



Plutôt que `cout`, on utilise dans le cas des erreurs le flux standard d'erreur nommé `cerr`. Il s'utilise exactement de la même manière que `cout`. On peut ainsi séparer les informations qui doivent s'afficher dans la console et les informations qui sont dues à des erreurs.

Cette manière de faire est correcte. Cependant, cela ressemble un peu au mauvais exemple numéro 2 ci-dessus. 🤔 En effet, la fonction peut potentiellement écrire dans la console alors que ce n'est pas son rôle. De plus le programme continue, alors qu'une erreur est survenue. Le mieux à faire serait alors de lancer l'exception dans la fonction et de récupérer l'erreur, si elle se produit, dans le `main`. De cette manière, celui qui appelle la fonction a conscience qu'une erreur s'est produite.

Code : C++ - La meilleure solution

```

int division(int a,int b) // Calcule a divisé par b.
{
    if(b==0)
        throw string("ERREUR : Division par zéro !");
    else
        return a/b;
}

int main()
{
    int a,b;
    cout << "Valeur pour a : ";
    cin >> a;
    cout << "Valeur pour b : ";
    cin >> b;

    try
    {
        cout << a << " / " << b << " = " << division(a,b) << endl;
    }
    catch(string const& chaine)
    {
        cerr << chaine << endl;
    }
    return 0;
}

```

Vous pouvez remarquer que le **throw** ne se trouve pas directement à l'intérieur du bloc **try**, mais qu'il se trouve à l'intérieur d'une fonction qui est appelée, elle, dans un bloc **try**.



Le **else** dans la fonction **division** n'est pas nécessaire puisque si l'exception est levée, le reste du code jusqu'au **catch** n'est pas exécuté.

Cette fois, le programme ne plante plus et la fonction n'a plus d'effet de bord. C'est la meilleure solution.

Les exceptions standards

Maintenant que l'on sait gérer les exceptions, la question principale est de savoir quel type d'objet lancer.

Je vous ai présenté avant la possibilité de lancer des exceptions de type entier ou string. Il est également possible de lancer un objet par exemple qui contiendrait plusieurs attributs comme :

- Une phrase décrivant l'erreur.
- Le numéro de l'erreur.
- Le niveau de l'erreur (erreur fatale, erreur mineure...).
- L'heure à laquelle l'erreur est survenue.
- ...

Un bon moyen de réaliser ceci est de dériver la classe `exception` de la bibliothèque standard du C++. Eh oui, là aussi la SL vient à notre secours. 😊



On parle d'exception et pas d'erreur, puisque si on la traite, ce n'est plus une erreur. 😊

La classe exception

La classe `exception` est la classe de base de toutes les exceptions lancées par la bibliothèque standard. Elle est aussi spécialement pensée pour qu'on puisse la dériver afin de réaliser notre propre type d'exception. La définition de cette classe est :

Code : C++ - La classe exception

```

class exception
{
public:
    exception() throw() { }           // Constructeur.
    virtual ~exception() throw();     // Destructeur.

    virtual const char* what() const throw(); // Renvoie une chaîne
    "à la C" contenant des infos sur l'erreur.
};

```

Pour l'utiliser, il faut inclure le fichier d'en-tête correspondant, soit ici le fichier `exception`.



Vous pouvez remarquer que la classe possède des fonctions virtuelles et donc également un destructeur virtuel. C'est un bon exemple de polymorphisme.



Les méthodes de la classe sont suivies du mot-clé **throw**. Cela sert à indiquer que ces méthodes ne vont pas lancer d'exceptions... ce qui est bien parce que si la classe d'exception commence à lancer des exceptions, on n'est pas sorti de l'auberge. 😊

Indiquer qu'une méthode ne lance pas d'exception est un mécanisme du C++ très rarement utilisé. En fait, cette classe est à peu près le seul endroit où vous verrez cela.

On peut alors créer sa propre classe d'exception en la dérivant grâce à un héritage. Ce qui donnerait par exemple :

Code : C++

```

#include <exception>
using namespace std;

class Erreur: public exception
{
public:
    Erreur(int numero=0, string const& phrase="", int niveau=0)
    throw()
        :m_numero(numero), m_phrase(phrase), m_niveau(niveau)
    {}

    virtual const char* what() const throw()
    {
        return m_phrase.c_str();
    }

    int getNiveau() const throw()
    {
        return m_niveau;
    }

    virtual ~Erreur() throw()
    {}

private:
    int m_numero;           // Numéro de l'erreur.
    string m_phrase;        // Description de l'erreur.
    int m_niveau;           // Niveau de l'erreur.
};

```

On pourrait alors récrire notre fonction de division de 2 entiers de la manière suivante :

Code : C++

```

int division(int a,int b) // Calcule a divisé par b.
{
    if(b==0)
        throw Erreur(1,"Division par zéro",2);
    else
        return a/b;
}

int main()
{
    int a,b;
    cout << "Valeur pour a : ";
    cin >> a;
    cout << "Valeur pour b : ";
    cin >> b;

    try
    {
        cout << a << " / " << b << " = " << division(a,b) << endl;
    }
    catch(std::exception const& e)
    {
        cerr << "ERREUR : " << e.what() << endl;
    }

    return 0;
}

```

Ce qui donne à l'exécution :

Code : Console

```

Valeur pour a : 3
Valeur pour b : 0
ERREUR : Division par zéro

```



Quel est l'intérêt de dériver la classe `exception`, alors qu'on pourrait faire sa propre classe sans aucun héritage ?

Excellente question. Il faut savoir que vous n'êtes pas le seul à lancer des exceptions. 😬 Certaines fonctions standards lancent elles aussi des exceptions. Toutes les exceptions lancées par les fonctions standards dérivent de la classe `exception`, ce qui permet avec un code générique de rattraper toutes les erreurs qui pourraient potentiellement arriver. Ce code générique est le suivant :

Code : C++

```

catch(std::exception const& e)
{
    cerr << "ERREUR : " << e.what() << endl;
}

```

Ceci est possible grâce au polymorphisme. On attrape un objet de type `exception`, mais grâce aux fonctions virtuelles et à la référence (les deux ingrédients !), c'est la méthode `what()` de la classe fille qui sera appelée, ce qui est justement ce que l'on souhaite. 😊

La bibliothèque standard peut lancer 5 types d'exceptions différents résumés dans le tableau suivant :

Nom de la classe	Description
<code>bad_alloc</code>	Lancée s'il se produit une erreur lors d'une manipulation de la mémoire.
<code>bad_cast</code>	Lancée s'il se produit une erreur lors d'un dynamic_cast .
<code>bad_exception</code>	Lancée si aucun catch ne correspond à un objet lancé.
<code>bad_typeid</code>	Lancée s'il se produit une erreur lors d'un typeid .
<code>ios_base::failure</code>	Lancée s'il se produit une erreur avec un flux.

On peut par exemple observer un exemple de `bad_alloc` avec le code suivant :

Code : C++

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    try
    {
        vector<int> a(1000000000,1);    //Un tableau bien trop grand
    }
    catch(exception const& e)          // On rattrape les exceptions
    standards de tous types.
    {
        cerr << "ERREUR : " << e.what() << endl;    // On affiche la
    description de l'erreur.
    }
    return 0;
}
```

Ce qui donne le résultat suivant dans la console :

Code : Console

```
ERREUR : std::bad_alloc
```



Si l'on avait attrapé l'exception par valeur et pas par référence (c'est-à-dire sans le `&`), le message aurait été `std::exception`, car le polymorphisme n'est pas conservé. C'est pour cela que l'on attrape toujours les exceptions par référence. C'est fort quand même ce polymorphisme ! 🤔

Le travail pré-mâché

Si comme moi (et beaucoup de programmeurs 🤔) vous êtes un fainéant et que vous n'avez pas envie de créer votre propre classe d'exception, sachez qu'il existe un fichier standard qui contient des classes d'exception pour les cas les plus courants. Le fichier `stdexcept` contient 9 classes d'exceptions séparées en 2 catégories, les exceptions « logiques » (*logic errors* en anglais) et les exceptions « d'exécution » (*runtime errors* en anglais).

Toutes les exceptions présentées dérivent de la classe `exception` et possèdent un constructeur prenant en argument une chaîne de caractère permettant de décrire le problème.

Nom de la classe	Catégorie	Description
<code>domain_error</code>	logique	A Lancer s'il se produit une erreur de domaine mathématique.
<code>invalid_argument</code>	logique	A Lancer si un des arguments d'une fonction est invalide.
<code>length_error</code>	logique	A Lancer si un objet aura une taille invalide. Par exemple si la classe Pile vue précédemment a une taille dépassant la taille de la mémoire.
<code>out_of_range</code>	logique	A Lancer s'il y a une erreur avec un indice. Par exemple si on essaye d'accéder à une case inexistante d'un tableau.
<code>logic_error</code>	logique	A Lancer lors de n'importe quel autre problème de logique du programme.
<code>range_error</code>	exécution	A Lancer lors d'une erreur de domaine à l'exécution.
<code>overflow_error</code>	exécution	A Lancer s'il y a une erreur d' overflow .
<code>underflow_error</code>	exécution	A Lancer s'il y a une erreur d' underflow .
<code>runtime_error</code>	exécution	A Lancer pour tout autre type d'erreur non-prévue survenant à l'exécution.

Si vous ne savez pas quoi choisir, prenez simplement `runtime_error`, cela n'a de toute façon que peu d'importance.



Et comment on les utilise ?

Reprenons une dernière fois notre exemple de division. Nous avons une erreur de domaine mathématique si l'argument `b` est nul. Choisissons donc de lancer une `domain_error`.

Code : C++

```
int division(int a,int b) // Calcule a divisé par b.
{
    if(b==0)
        throw domain_error("Division par zéro");
    else
        return a/b;
}
```



On aurait très bien pu choisir une `argument_error` ou encore une `runtime_error`. Cela n'a que peu d'importance puisqu'en général on attrape toujours les exceptions par la méthode indiquée plus haut.

Les exceptions de vector

Je vous ai dit dans l'introduction qu'une erreur possible (et courante !) était le cas où un utilisateur cherche à accéder à la 10^{ème} case d'un `vector` de 8 éléments.

Accéder aux objets stockés dans un tableau, vous savez le faire depuis longtemps. On utilise bien sûr les crochets `[]`. Or, ces crochets ne font aucun test. Si vous fournissez un index invalide, le programme va planter et c'est tout. 😞

Et après ce chapitre, on pourrait se demander si c'est vraiment une bonne idée. Utiliser une exception en cas d'erreur d'index vous paraît peut-être une bonne idée... et aux concepteurs de la STL aussi ! 😊

C'est pour ça que les `vector` (et les `deque`) proposent une méthode appelée `at()` qui fait exactement la même chose que les crochets mais qui lance une exception en cas d'indice erroné.

Code : C++

```
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<double> tab(5, 3.14); //Un tableau de 5 nombres à virgule

    try
    {
        tab.at(8) = 4.; //On essaye de modifier la 8ème case
    }
    catch(exception const& e)
    {
        cerr << "ERREUR : " << e.what() << endl;
    }
    return 0;
}
```

Ce qui nous donne :

Code : Console

```
ERREUR : vector::_M_range_check
```

Encore un nouveau type d'exception ! 😞 Oui, oui, mais ce n'est pas grave. Car comme je vous l'ai dit, tous les types d'exception utilisés dérivent de la classe `exception` et notre `catch` "standard" est donc suffisant. Il n'y a donc qu'une seule syntaxe à apprendre. Plutôt sympa non ? 😎



En pratique, on utilise très rarement, voire même jamais la méthode `at()`. On considère plutôt que c'est à l'utilisateur de `vector` d'utiliser le tableau correctement.

Terminons avec un point qui pourrait vous sauver la vie lors de la lecture de codes sources obscures.

Relancer une exception

Il est possible de relancer une exception reçue par un bloc `catch` afin de la retraiter une deuxième fois plus loin dans le code. Pour ce faire, il faut utiliser le mot-clé `throw` sans expression derrière.

Code : C++

```
catch(exception const& e) // Rattrape toutes les exceptions
{
    //On traite une première fois l'exception
    cerr << "ERREUR: " << e.what() << endl;

    throw; // Et on relance l'exception reçue pour la retraiter
           // dans un autre bloc catch plus loin dans le code.
}
```

Les assertions

Les exceptions c'est bien. Mais il y a des cas où mettre en place tous ces blocs `try` / `catch` est fastidieux. Ce n'est pas pour rien que `vector` propose les `[]` pour accéder aux éléments. On n'a pas toujours envie d'avoir à traiter les exceptions. Il existe un autre mécanisme de détection et gestion qui vient du langage C : les assertions.

Claquer une assertion

Pour utiliser les assertions, il faut inclure le fichier d'en-tête `cassert`. Et c'est certainement l'étape la plus difficile. 😊

Une assertion permet de tester si une expression est vraie ou non. Si c'est vrai, rien ne se passe et le programme continue. Par contre, si le teste est négatif, le programme s'arrête brutalement et un message d'erreur s'affiche dans le terminal.

Code : C++

```
#include <cassert>
using namespace std;

int main()
{
    int a(5);
    int b(5);

    assert(a == b) ; //On vérifie que a et b sont égaux

    //reste du programme
    return 0;
}
```

Lors de l'exécution, rien ne se passe, normal les deux variables sont égales. 😊 Par contre, si vous modifiez la valeur de `b`, alors le message suivant s'affiche à l'exécution :

Code : Console

```
monProg: main.cpp:9: int main(): Assertion `a == b' failed.
Abandon
```

C'est super, le message d'erreur indique le fichier où se situe l'erreur, le nom de la fonction et même la ligne ! Avec ça, impossible de ne pas trouver la cause d'erreur. Je vous avais bien dit que c'était simple !



Mais pourquoi utiliser des exceptions si les assertions sont mieux ?

Attention, je n'ai pas dit que les assertions étaient mieux ! Les deux méthodes de gestion des erreurs ont leur domaine d'application. Si vous claquez une assertion, le programme s'arrête brutalement. Il n'y a aucun moyen de réparer l'erreur et tenter de continuer. Si vous avez un programme de chat et qu'il n'arrive pas à se connecter au serveur, c'est une erreur. Vous aimeriez bien que votre programme réessaye de se connecter plusieurs fois. Il faut donc utiliser une exception, pour tenter de réparer l'erreur. Une assertion aurait complètement tué le programme. Ce n'est clairement pas la bonne solution dans ce cas !

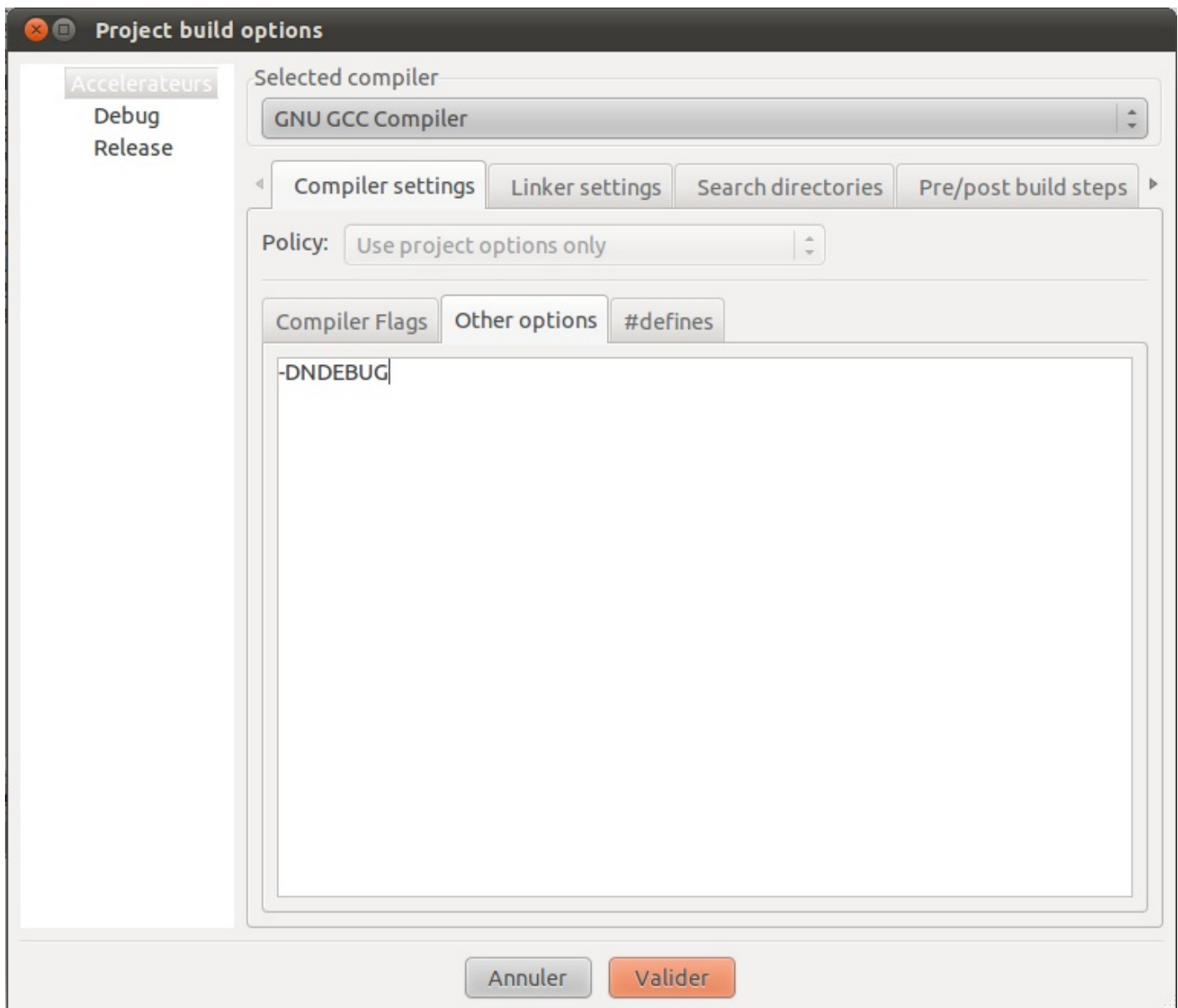
A vous de choisir ce dont vous avez besoin au cas par cas.

Désactiver les assertions

Un autre point fort des assertions est la possibilité de les **désactiver totalement**. En faisant ça, le compilateur va simplement ignorer les lignes `assert (. . .)` et ne pas effectuer le test qui se trouve entre les parenthèses. En faisant ça, le code sera (légèrement) plus rapide, mais aucun test ne sera effectué. Il faut donc choisir. 🤖

Pour désactiver les assertions, il faut ajouter l'option `-DNDEBUG` à la ligne de compilation.

Si vous utilisez Code::Blocks, cela se fait via le menu *project > build options*. Dans la fenêtre qui s'ouvre il faut sélectionner l'onglet *Compiler settings* et dans le champ *Other options*, vous ajoutez simplement `-DNDEBUG`, comme sur l'illustration suivante.



Avec cette option activée, le code d'exemple précédent s'exécute sans problème même si *a* est différent de *b*. La ligne de test a simplement été ignorée par le compilateur.



Les assertions sont souvent utilisées durant la phase de création d'un programme pour tester si tout se passe bien. Une fois que l'on sait que le programme fonctionne, on les désactive, on compile et on vend le programme au client. 😊 Ce dernier ne veut pas de messages d'erreurs et il veut un programme rapide. Si par contre il découvre un bug, on réactive les assertions et on cherche l'erreur. C'est vraiment un outil destiné aux développeurs, au contraire des exceptions.

Armés de ces nouveaux outils, vous êtes maintenant aptes à créer des projets plus robustes et à gérer les événements inattendus qui pourraient surgir durant l'exécution de vos futurs programmes.

Sachez tout de même que la gestion *correcte* de toutes les exceptions qui pourraient survenir dans un programme est une tâche qui peut s'avérer très compliquée. Penser à tout est souvent bien plus difficile que l'on se l'imagine. 😎

Créer des templates

Revenons un peu en arrière et réfléchissons aux raisons pour lesquelles on cherche à programmer. Le but de la programmation, en tout cas à l'origine, est de simplifier les tâches répétitives en les faisant s'exécuter sur votre ordinateur plutôt que devoir faire tous les calculs à la main. On veut donc s'éviter du travail à la chaîne. Et cela reste valide pour le programmeur : s'il peut réutiliser un bout de code plutôt que de devoir le réécrire, alors il gagne du temps. 😊

Dans les chapitres sur le polymorphisme, nous avons vu un moyen d'exécuter un code différent pour deux types semblables. Cette fois, nous allons voir comment faire s'exécuter un même code pour différents types de variables ou classes. Cela nous permettra d'éviter la tâche répétitive de réécriture de portions de code semblables pour différents types. Pensez à la classe `vector`. Quel que soit le type d'objet que l'on stocke, le tableau aura le même comportement : ajouter et supprimer des éléments, renvoyer sa taille, etc. Finalement, peu importe que ce soit un tableau d'entiers ou de nombres réels.

La force des *templates* est d'autoriser une fonction ou une classe à utiliser des types différents. C'est un mécanisme unique au C++ que l'on ne trouve que dans peu d'autres langages. La marque de fabrique des templates sont les chevrons `< >` et vous l'aurez remarqué, la STL utilise énormément ce concept.



L'utilisation des templates est un sujet très vaste. Il y a même des livres entiers qui y sont consacrés tellement cela peut devenir complexe mais néanmoins puissant. Ce chapitre n'est qu'une brève introduction au domaine.

Les fonctions templates Ce que l'on aimerait faire

Il arrive souvent qu'on ait besoin d'opérations mathématiques dans nos programmes. Une opération toute simple est celle qui consiste à trouver le plus grand de deux nombres. Dans le cas des nombres entiers, on pourrait écrire une fonction comme suit :

Code : C++ - La fonction maximum

```
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```



Une telle fonction existe bien sûr dans la SL. Elle se trouve dans l'en-tête `algorithm` et s'appelle simplement `max()`

Cette fonction est très bien et elle n'a pas de problème. Cependant, si un utilisateur de votre fonction aimerait utiliser des `double` à la place des `int`, il risque d'avoir un problème. Il faudrait donc fournir également une version de cette fonction utilisant des nombres réels. Ce qui ne devrait pas vous poser de problème à ce stade du cours. 😊

Pour être rigoureux, il faudrait également fournir une fonction de ce type pour les `char`, les `unsigned int` les nombres rationnels, etc. On se rend vite compte que la tâche est très répétitive.

Cependant, il y a un point commun à toutes ces fonctions, le **corps de la fonction est strictement identique**. Quel que soit le type, le traitement que l'on effectue est le même. On se rend compte que l'algorithme utilisé dans la fonction est *générique*.

Il serait donc intéressant de pouvoir écrire une seule fois la fonction en disant au compilateur : « Cette fonction est la même pour tous les types, fais le sale boulot de recopie du code toi-même. » Eh bien, ça tombe bien parce que c'est ce que permettent les *templates* en C++ et c'est ce que nous allons apprendre à utiliser dans la suite.



Le terme français pour *template* est *modèle*. Le nom est bien choisi car il décrit précisément ce que nous allons faire. Nous allons écrire un modèle de fonction et le compilateur va utiliser ce modèle dans les différents cas qui nous intéressent.

Une première fonction template

Pour indiquer au compilateur que l'on veut faire une fonction générique, on va déclarer un « type variable » qui peut représenter n'importe quel autre type. On parle de type générique. Cela se fait de la manière suivante :

Code : C++ - Déclaration d'un type générique

```
template<typename T>
```

Vous pouvez remarquer quatre choses importantes.

1. Premièrement le mot-clé **template** qui prévient le compilateur que la prochaine chose dont on va lui parler sera générique.
2. Deuxièmement, les symboles "<" et ">" que vous avez certainement déjà aperçus dans le chapitre sur les `vector` et sur la SL. C'est la marque de fabrique des templates.
3. Troisièmement, le mot-clé **typename** qui indique au compilateur que **T** sera le nom que l'on va utiliser pour notre « type spécial » qui remplace n'importe quoi.
4. Finalement, il n'y a **PAS** de point-virgule à la fin de la ligne.



On peut également utiliser le mot-clé **class** à la place de **typename** dans ce contexte. Il n'y a *aucune* différence. Cela donne : **template<class T>**. J'utiliserai **typename** dans la suite pour éviter les confusions.

Secret (cliquez pour afficher)

Beaucoup de programmeurs utilisent **class** à la place de **typename** en invoquant la raison que cela fait 3 caractères de moins à taper... 🤪

La ligne de code précédente indique au compilateur que dans la suite, **T** sera un type générique pouvant représenter n'importe quel autre type. On pourra donc utiliser ce **T** dans notre fonction comme type pour les arguments et pour le type de retour.

Code : C++ - Ma première fonction template

```
template <typename T>
T maximum(const T& a, const T& b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

Quand il va voir cela, le compilateur va automatiquement générer une série de fonctions `maximum()` pour tous les types dont vous avez besoin. Cela veut dire que si vous avez besoin de cette fonction pour des entiers, le compilateur va créer la fonction :

Code : C++

```
int maximum(const int& a, const int& b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

... et de même pour les `double`, `char`, etc. C'est le compilateur qui se farcit le travail de recopie ! Parfait, on peut aller faire la sieste pendant ce temps. 🤪

On peut écrire un petit programme de test :

Code : C++ - Test de la fonction maximum

```
#include <iostream>
using namespace std;

template <typename T>
T maximum(const T& a, const T& b)
{
    if(a>b)
        return a;
    else
        return b;
}

int main()
{
    double pi(3.14);
    double e(2.71);

    cout << maximum<double>(pi,e) << endl; // Utilise la "version
double" de la fonction.

    int cave(-1);
    int dernierEtage(12);

    cout << maximum<int>(cave,dernierEtage) << endl; // Utilise
la "version int" de la fonction.

    unsigned int a(43);
    unsigned int b(87);

    cout << maximum<unsigned int>(a,b) << endl; // Utilise la
"version unsigned int" de la fonction.

    return 0;
}
```

Et tout cela se passe sans que l'on ait besoin d'écrire plus de code. Il faut juste indiquer entre des chevrons quelle "version" de la fonction on souhaite utiliser, comme pour les `vector` en somme : on devait indiquer quelle "version" du tableau on souhaitait utiliser.

Il n'est pas toujours utile d'indiquer entre chevrons quel type l'on souhaite utiliser pour les fonctions templates. Le compilateur est assez intelligent pour *deviner* ce que vous souhaitez faire. Mais dans des cas compliqués ou si il y a plusieurs arguments de types différents, alors il devient nécessaire de spécifier la version.

Code : C++

```
int main()
{
    double pi(3.14);
    double e(2.71);

    cout << maximum(pi,e) << endl; // Utilise la "version double"
de la fonction.
    return 0;
}
```

Le compilateur voit dans ce cas que l'on souhaite utiliser la "version `double`" de la fonction.
A vous de voir si votre compilateur comprend vos intentions. 🤪

Si vous êtes attentifs, vous avez peut-être remarqué que j'ai remplacé le passage par valeur pour les arguments par des [références constantes](#). En effet, on ne sait pas quel type l'utilisateur va utiliser avec notre fonction `maximum()`. La taille en mémoire de ce type sera peut-être très grande ; on passe donc une référence constante pour éviter une copie coûteuse inutile.

Où mettre la fonction ?

Habituellement, un programme est subdivisé en plusieurs fichiers que l'on classe en deux catégories. Les fichiers de code (les `.cpp`) et les fichiers d'en-tête (les `.h`). Généralement, on met le prototype de la fonction dans un `.h` et la définition dans le `.cpp` comme on l'a vu [tout au début de ce cours](#).

Pour les fonctions templates, c'est différent. TOUT doit obligatoirement se trouver dans le fichier `.h`, sinon votre programme ne pourra pas compiler.

Je le répète encore une fois, car c'est une erreur classique, **le prototype ET la définition d'une fonction template doivent obligatoirement se trouver dans un fichier d'en-tête.**

Tous les types sont-ils utilisables ?

J'ai dit plus haut que le compilateur allait générer toutes les fonctions nécessaires. Cependant, il y a quand même une contrainte ici : le type que l'on passe à la fonction doit posséder un **opérateur** `>`. Par exemple, on ne peut pas utiliser cette fonction avec un `Personnage` ou un `Magicien` des chapitres précédents : ils ne possèdent pas de surcharge de `>`. Tant mieux, puisque prendre le maximum de deux `Personnages` n'a pas de sens !

Les contraintes dépendent des fonctions que vous écrivez. Si vous utilisez l'opérateur `+` dans la fonction, alors il faut que l'objet passé en argument surcharge cet opérateur. Si vous effectuez une copie dans la fonction, alors l'objet doit posséder un constructeur de copie etc.

Des fonctions plus compliquées

Vous aviez appris à écrire une fonction qui calcule la moyenne d'un tableau. A nouveau, les opérations à effectuer sont les mêmes quel que soit le type contenu. Écrivons donc cette fonction sous forme de modèle template.

Voici ma version :

Code : C++ - Moyenne. Un nouvel espoir

```
template<typename T>
T moyenne(T tableau[], int taille)
{
    T somme(0); //La somme des éléments du tableau
    for(int i(0); i<taille; ++i)
        somme += tableau[i];

    return somme/taille;
}
```



Tous les arguments d'une fonction ne doivent pas forcément être templates. Ici, `taille` est un entier tout ce qu'il y a de plus normal dans toutes les versions de la fonction.

Le problème que nous avions était que pour le type `int`, on se retrouvait avec une division entière qui posait problème (Les moyennes étaient arrondies vers le bas). Ce problème serait résolu si l'on pouvait utiliser un type différent de `int` pour la somme et donc la moyenne.

Pas de problème. Ajoutons donc un deuxième paramètre template pour le type de retour et utilisons le.

Code : C++ - Moyenne 2. Le template contre-attaque


```
template<typename T, typename S>
S moyenne(T tableau[], int taille)
{
    S somme(0); //La somme des éléments du tableau
    for(int i(0); i<taille; ++i)
        somme += tableau[i];

    return somme/taille;
}
```

Avec cela, il est enfin possible de calculer la moyenne correctement. 😎

Par contre, il faut explicitement indiquer les types à utiliser lors de l'appel de la fonction. Le compilateur ne peut pas deviner quel type vous aimeriez pour S :

Code : C++ - Moyenne 3. Le retour du main

```
#include<iostream>
using namespace std;

template<typename T, typename S>
S moyenne(T tableau[], int taille)
{
    S somme(0); //La somme des éléments du tableau
    for(int i(0); i<taille; ++i)
        somme += tableau[i];

    return somme/taille;
}

int main()
{
    int tab[5];
    //Remplissage du tableau

    cout << "Moyenne : " << moyenne<int,double>(tab,5) << endl;

    return 0;
}
```

De cette manière, on peut spécifier le type utilisé pour le calcul de la moyenne tout en préservant la liberté totale sur le type contenu dans le tableau. Pour bien assimiler le tout, je ne peux que vous inviter à faire quelques exercices, par exemple :

- Écrire une fonction renvoyant le plus petit de deux éléments.
- Réécrire la fonction `moyenne()` pour qu'elle reçoive en argument un `std::vector<T>` à la place d'un tableau statique.
- Écrire une fonction template renvoyant un nombre aléatoire d'un type donné.

La spécialisation

Pour l'instant, nous n'avons essayé la fonction `maximum()` qu'avec des types de base. Essayons-la donc avec une chaîne de caractères :

Code : C++

```
int main()
{
    cout << "Le plus grand est: " <<
    maximum<std::string>("elephant", "souris") << endl;

    return 0;
}
```

Le résultat de ce petit programme est :

Code : Console

```
Le plus grand est: souris
```

On l'a déjà vu, l'opérateur < pour les chaînes de caractère compare selon l'ordre lexicographique. Mais imaginons (comme précédemment 🤔) que le critère de comparaison qui nous intéresse est la longueur de la chaîne. Cela se fait en *spécialisant* la fonction template.

La spécialisation

La spécialisation se fait en utilisant la syntaxe suivante :

Code : C++

```
template <>
string maximum<string>(const string& a, const string& b)
{
    if(a.size()>b.size())
        return a;
    else
        return b;
}
```

Vous remarquerez deux choses:

- La première ligne qui ne comporte *aucun* type entre < et >.
- Le prototype de la fonction qui utilise cette fois le type que l'on veut et plus le type générique T.

Avec cette spécialisation, on obtient le comportement voulu :

Code : C++

```
int main()
{
    cout << "Le plus grand est: " <<
    maximum<std::string>("elephant", "souris") << endl;

    return 0;
}
```

qui donne :

Code : Console

```
Le plus grand est: elephant
```

La seule difficulté de la spécialisation est la syntaxe qui commence par la ligne `template<>`. Si vous vous souvenez de ça, vous savez tout.



Vous pouvez évidemment spécialiser la fonction pour plusieurs types différents. Il vous faudra alors créer une



spécialisation par type.

L'ordre des fonctions

Pour pouvoir compiler et avoir le comportement voulu, votre programme devra être organisé d'une manière spéciale. Il faut respecter un ordre particulier :

1. La fonction générique
2. Les fonctions spécialisées

L'ordre est essentiel.

Lors de la compilation, le compilateur cherche une fonction spécialisée. S'il n'en trouve pas, alors il utilise la fonction générique déclarée au-dessus.

Les classes templates

Voyons maintenant comment réaliser des classes template, c'est-à-dire des classes dont le type des arguments peut varier. Cela peut vous sembler effrayant, mais vous en avez déjà utilisé beaucoup. Pensez à `vector` ou `deque` par exemple.

Il est temps de savoir réaliser des modèles de classes utilisables avec différents types.

Je vous propose de travailler sur un exemple que l'on pourrait trouver dans une bibliothèque comme Qt. Lorsque l'on veut dessiner des choses à l'écran, on utilise quelques formes de base qui servent à décomposer les objets plus complexes. L'une de ces formes est le rectangle qui comme vous l'aurez certainement remarqué est la forme des fenêtres ou des boutons entre autres.



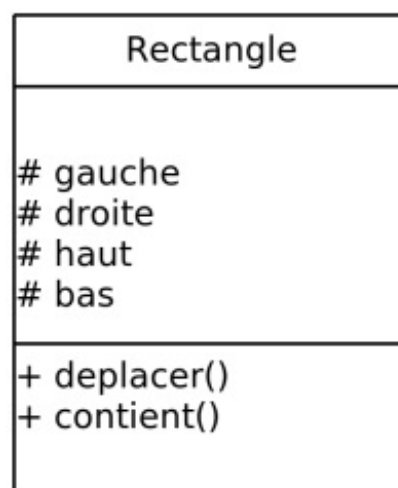
Quelles sont les propriétés d'un rectangle ?

Un rectangle a quatre côtés, une surface et un périmètre. Les deux derniers éléments peuvent être calculés si l'on connaît les quatre arêtes. Voilà pour les attributs.

Quelles sont les actions qu'un rectangle peut effectuer ?

Ici, il y a beaucoup de choix. Nous choisirons donc les actions suivantes : vérifier si un point est contenu dans le rectangle et déplacer le rectangle.

Nous pourrions donc modéliser notre classe de la sorte :



On considère ici un rectangle parallèle aux bords de l'écran ce qui permet de simplifier les positions en utilisant un seul et unique nombre par côté.

Le type des attributs

Maintenant que nous avons modélisé la classe, il est temps de réfléchir aux types des attributs, en l'occurrence la position des côtés.

Si l'on veut avoir une bonne précision, alors il nous faut utiliser des `double` ou des `float`. Si par contre on considère que de toute façon l'écran est composé de pixels, on peut se dire que l'utilisation d'`int` est largement suffisante.

Les deux options sont possibles et on peut très bien avoir besoin des deux approches dans un seul et même programme. Et c'est là que vous devriez tous me dire : "Mais alors, utilisons donc des templates !". Vous avez bien raison. Nous allons écrire une seule classe qui pourra être instanciée avec différents types par le compilateur.

Création de la classe

Je suis sûr que vous connaissez la syntaxe même si je ne vous l'ai pas encore donnée. 🤖 Comme d'habitude, on déclare un type générique `T`. Puis on déclare notre classe.

Code : C++ - La syntaxe

```
template <typename T>
class Rectangle{
    //...
};
```

Notre type générique est reconnu par le compilateur à l'intérieur de la classe. Utilisons-le donc pour déclarer nos quatre attributs.

Code : C++ - Les attributs

```
template <typename T>
class Rectangle{

    //...

private:

    //Les côtés du Rectangle
    T m_gauche;
    T m_droite;
    T m_haut;
    T m_bas;

};
```

Voilà. Jusque-là, ce n'était pas bien difficile. Il ne nous reste plus qu'à écrire les méthodes. 🤖

Les méthodes

Les fonctions les plus simples à écrire sont certainement les accesseurs qui permettent de connaître la valeur des attributs. La hauteur d'un rectangle est évidemment la différence entre la position du haut et la position du bas. Comme vous vous en doutez, cette fonction est template puisque le type de retour de la fonction sera un `T`.

Une première méthode

Nous pouvons donc écrire la méthode suivante :

Code : C++ - Première fonction membre

```
template <typename T>
```

```

class Rectangle{
public:
    //...

    T hauteur() const
    {
        return m_haut-m_bas;
    }

private:
    //Les cotes du Rectangle
    T m_gauche;
    T m_droite;
    T m_haut;
    T m_bas;
};

```

Vous remarquerez qu'il n'y a pas besoin de redéclarer le type template **T** juste avant la fonction membre puisque celui que nous avons déclaré avant la classe reste valable pour tout ce qui se trouve à l'intérieur.



Et si je veux mettre le corps de ma fonction à l'extérieur de ma classe ?

Bonne question. On prend souvent l'habitude de séparer le prototype de la définition. Et cela peut se faire aussi ici. Pour faire cela, on mettra le prototype dans la classe et la définition à l'extérieur mais il faut ré-indiquer qu'on utilise un type variable **T** :

Code : C++ - Fonction membre à l'extérieur

```

template <typename T>
class Rectangle{
public:
    //...

    T hauteur() const;

    //...
};

template<typename T>
T Rectangle<T>::hauteur() const
{
    return m_haut-m_bas;
}

```

Vous remarquerez aussi l'utilisation du type template dans le nom de la classe puisque cette fonction sera instanciée de manière différente pour chaque **T**.



Souvenez-vous que tout doit se trouver dans le fichier .h !

Une fonction un peu plus complexe

Une des fonctions que nous voulions écrire est celle permettant de vérifier si un point est contenu dans le rectangle ou pas. Pour cela, on doit passer un point (x, y) en argument à la fonction. Le type de ces arguments doit évidemment être **T**, de sorte que l'on puisse comparer les coordonnées sans avoir de conversions.

Code : C++

```

template <typename T>
class Rectangle{
public:

    //...

    bool estContenu(T x, T y) const
    {
        return (x >= m_gauche) && (x <= m_droite) && (y >= m_bas) &&
        (y <= m_haut);
    }

private:
    //...
};

```

Vous remarquerez à nouveau l'absence de redéfinition du type **T**. Quoi, je me répète ? 😊 C'est sûrement que cela devient clair pour vous.

Constructeur

Il ne nous reste plus qu'à traiter le cas du constructeur. A nouveau, rien de bien compliqué, on utilise simplement le type **T** défini avant la classe.

Code : C++

```

template <typename T>
class Rectangle{
public:

    Rectangle(T gauche, T droite, T haut, T bas)
        :m_gauche(gauche),
        m_droite(droite),
        m_haut(haut),
        m_bas(bas)
    {}

    //...
};

```

Et comme pour toutes les autres méthodes, on peut définir le constructeur à l'extérieur de la classe. Vous êtes bientôt des pros, je vous laisse donc essayer seuls.



On pourrait ajouter une fonction appelée dans le constructeur qui vérifie que le haut se trouve bien au-dessus du bas et de même pour droite et gauche.

Finalement, voyons comment utiliser cette classe.

Instanciation d'une classe template

Il fallait bien y arriver un jour ! Comment crée-t-on un objet d'une classe template et en particulier de notre classe Rectangle ?

En fait, je suis sûr que vous le savez déjà. 😊 Cela fait longtemps que vous créez des objets à partir de la classe template `vector` ou `map`. Si l'on veut un `Rectangle` composé de `double`, on devra écrire :

Code : C++

```
int main()
{
    Rectangle<double> monRectangle(1.0, 4.5, 3.1, 5.2);

    return 0;
}
```

L'utilisation des fonctions se fait ensuite comme d'habitude :

Code : C++

```
int main()
{
    Rectangle<double> monRectangle(1.0, 4.5, 3.1, 5.2);

    cout << monRectangle.hauteur() << endl;

    return 0;
}
```

Pour terminer ce chapitre, je vous propose d'ajouter quelques méthodes à cette classe. Je vous parlais d'une méthode `deplacer()` qui change la position du rectangle. Essayez autrement d'écrire les méthodes `surface()` et `perimetre()`.

Finalement, pour bien tester tout ces concepts, vous pouvez refaire la classe [ZFraction](#) de sorte à ce que l'on puisse spécifier le type à utiliser pour stocker le numérateur et le dénominateur. Bonne chance !

C'est un des chapitres les plus avancés de ce cours... mais certainement un des plus intéressants. 😊 Vous avez appris à faire travailler votre compilateur pour qu'il crée différentes versions d'une même fonction ou d'une même classe.

Ce chapitre n'est par contre qu'un bref aperçu. Il y aurait encore énormément de choses à dire sur le sujet, mais je suis convaincu que je vous ai donné envie d'en savoir plus dans le domaine.

Ce que vous pouvez encore apprendre

Qu'on se le dise : bien que le tutoriel C++ s'arrête là, vous ne savez pas tout sur tout. D'ailleurs, personne ne peut vraiment prétendre tout savoir sur le C++ et toutes ses bibliothèques.

L'objectif n'est pas de *tout savoir*, mais d'être capable d'apprendre ce dont vous avez besoin lorsque c'est nécessaire.

Si je devais moi-même vous apprendre tout sur le C++, j'y passerais toute une vie (et encore, ça serait toujours incomplet). J'ai autre chose à faire, et j'en serais de toute façon incapable.

Du coup, plutôt que de tout vous apprendre, j'ai choisi de vous enseigner de bonnes bases tout au long du cours. Cette annexe a pour but, maintenant que le cours est fini, de vous donner un certain nombre de pistes pour continuer votre apprentissage. 😊

J'ai découpé ce chapitre en 3 parties :

- Ce que vous pouvez encore apprendre sur le langage C++ lui-même.
- Ce que vous pouvez encore apprendre sur la bibliothèque Qt.
- Présentation de quelques autres bibliothèques. Il n'y a pas que Qt !



Cette annexe est seulement là pour vous **présenter** de nouvelles notions, pas pour vous les expliquer. Ne soyez donc pas surpris si je suis beaucoup plus succinct que d'habitude. Imaginez cette annexe comme un sommaire de ce qu'il vous reste à apprendre. 😊

... sur le langage C++

Le langage C++ est suffisamment riche pour qu'il vous reste encore de nombreuses notions à découvrir. Certaines d'entre elles sont particulièrement complexes, je ne vous le cache pas, et vous n'en aurez pas besoin tout le temps.

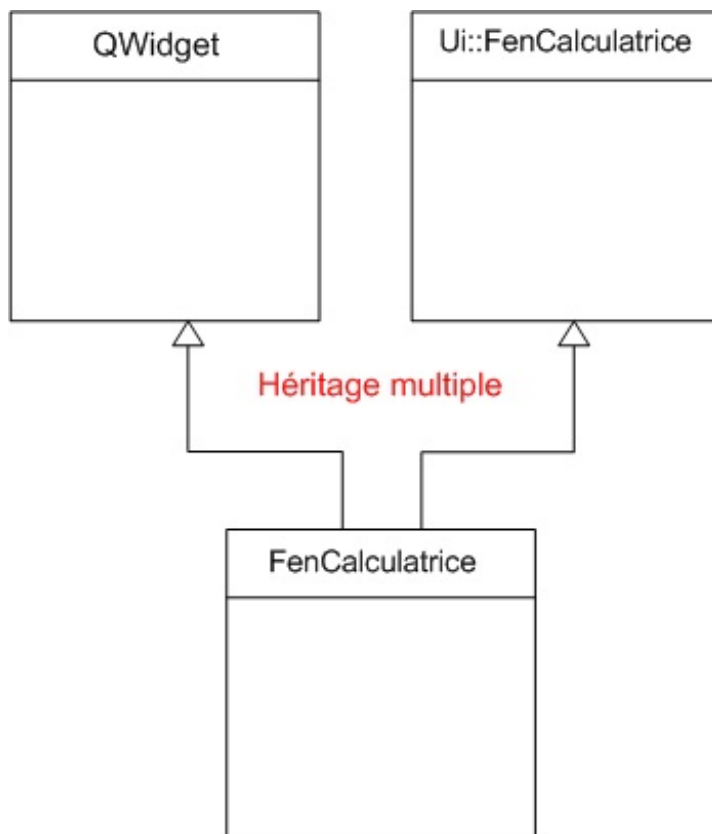
Toutefois, au cas où vous en ayez besoin un jour, je vais vous présenter *rapidement* ces notions. A vous ensuite d'approfondir vos connaissances, par exemple en lisant des [tutoriels écrits par d'autres Zéros sur le C++](#), en lisant des livres dédiés au C++, ou tout simplement en faisant une recherche Google. 😊

Voici les notions que je vais vous présenter ici :

- L'héritage multiple
- Les espaces de nom
- Les types énumérés
- Les **typedef**

L'héritage multiple

L'héritage multiple consiste à hériter de plusieurs classes à la fois. Nous avons déjà fait cela dans la partie sur Qt, pour pouvoir utiliser une interface dessinée dans Qt Designer :



Pour hériter de plusieurs classes, il suffit de mettre une virgule entre les noms de classe, comme on l'avait fait :

Code : C++

```
class FenCalculatrice : public QWidget, private Ui::FenCalculatrice
{
};
```

C'est une notion qui paraît simple mais qui, en réalité, est très complexe.

En fait, la plupart des langages de programmation plus récents, comme Java et Ruby, ont carrément décidé de ne pas gérer l'héritage multiple. Pourquoi ? Parce que ça peut être utile dans certaines conditions assez rares, mais si on l'utilise mal (quand on débute) ça peut devenir un cauchemar à gérer.

Bref, jetez un coup d'oeil à cette notion, mais juste un coup d'oeil de préférence, car vous ne devriez pas y avoir recours souvent.



Les namespaces

Souvenez-vous. Dès le début du tutoriel C++, je vous ai fait utiliser les objets `cout` et `cin` qui permettent d'afficher un message dans la console et de récupérer le texte saisi au clavier.

Voici le tout premier code source C++ que vous aviez découvert mais avec le vrai nom des objets :

Code : C++

```
#include <iostream>

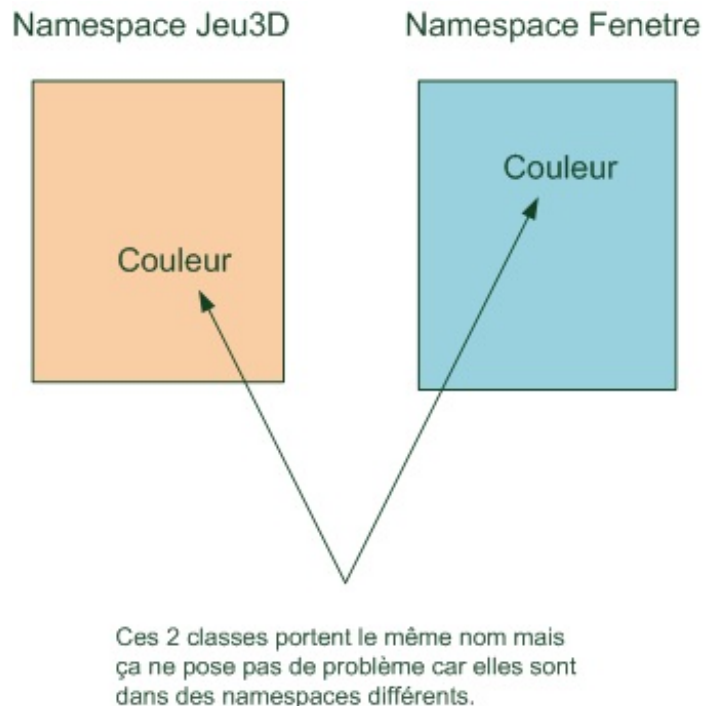
int main()
{
    std::cout << "Hello world!" << std::endl;
```

```
        return 0;  
    }
```

Le préfixe "std::" correspond à ce qu'on appelle un **namespace**, c'est-à-dire *espace de nom* en français. Les namespaces sont utiles dans de très gros programmes où il y a beaucoup de noms de classes et de variables différents.

Quand vous avez beaucoup de noms différents dans un programme, il y a un risque que 2 classes aient le même nom. Par exemple, vous pourriez utiliser 2 classes `Couleur` dans votre programme : une dans votre bibliothèque "Jeu3D" et une autre dans votre bibliothèque "Fenetre".

Normalement, avoir 2 classes du même nom est interdit... sauf si ces classes sont chacune dans un namespace différent ! Imaginez que les namespaces sont comme des "boîtes" qui évitent de mélanger les noms de classes et de variables.



Si la classe est dans un namespace, on doit placer le nom du namespace en préfixe devant :

Code : C++

```
Jeu3D::Couleur rouge; // Utilisation de la classe Couleur située  
dans le namespace Jeu3D  
Fenetre::Couleur vert; // Utilisation d'une AUTRE classe appelée  
elle aussi Couleur, dans le namespace Fenetre
```

Les espaces de noms sont vraiment comme des noms de famille pour les noms de variables.

Le namespace "std" est utilisé par toute la bibliothèque standard du C++. Il faut donc mettre ce préfixe devant chaque nom issu de la bibliothèque standard (`cout`, `cin`, `vector`, `string`...).

Il est aussi possible, comme on le fait depuis le début, d'utiliser la directive **using namespace** au début du fichier :

Code : C++

```
using namespace std;
```

Grâce à ça, dans tout le fichier le compilateur saura que vous faites références à des noms définis dans l'espace de nom `std`.

Cela vous évite d'avoir à répéter `std::` partout.



Certains programmeurs préfèrent éviter d'utiliser "using namespace" car, en lisant le code ensuite, on ne sait plus vraiment à quel namespace le nom se rapporte.

Les types énumérés

Dans nos programmes, on a parfois besoin de manipuler des variables qui ne peuvent prendre qu'un petit nombre de valeurs différentes. 😊 Tenez, si vous devez décrire les trois niveaux de difficulté de votre jeu, vous pourriez utiliser un `int` valant 1, 2 ou 3. Mais ce n'est pas très sécurisé, on n'est pas sûr que notre entier prendra toujours une de ces trois valeurs. Il serait bien d'avoir un type qui ne peut prendre *que* ces trois valeurs.

Un type énuméré se déclare comme ceci :

Code : C++

```
enum Niveau{Facile, Moyen, Difficile};
```

On l'utilise alors comme n'importe quelle autre variable.

Code : C++

```
int main()
{
    Niveau level;

    //...

    if(level == Moyen)
        cout << "Vous avez choisi le niveau moyen" << endl;

    //...

    return 0;
}
```

C'est bien pratique. Et en plus, cela rend le code plus lisible. La ligne `if(level == Moyen)` est plus claire à lire que `if(level == 2)`. On n'a pas besoin de réfléchir à ce que représente ce 2.

On retrouve souvent les types énumérés dans des codes utilisant les tests `switch`. Voici un exemple utilisant un type énuméré pour les directions d'un personnage sur une carte :

Code : C++

```
enum Direction{Nord, Sud, Est, Ouest};

int main()
{
    Direction dir;
    Personnage p;

    //...

    switch(dir)
    {
        case Nord: p.avancerNord(); break;
        case Sud: p.avancerSud(); break;
    }
}
```

```

        case Est: p.avancerEst(); break;
        case Ouest: p.avancerOuest(); break;
    }

    //...

    return 0;
}

```

Certains programmeurs utilisent des **enum** partout alors que d'autres n'aiment pas. Faites comme vous préférez. 😊

Les typedefs

Vous en voulez encore ? Voyons donc une petite astuce bien pratique pour économiser du texte.

Certains types sont vraiment long à écrire. Prenez par exemple un itérateur sur une table associative de chaînes de caractères et de vector d'entiers. Un objet de ce type se déclare comme ceci:

Code : C++

```
std::map<std::string, std::vector<int> >::iterator it;
```

C'est un peu long ! 😞

Les typedefs (redéfinition de type en français) permettent de créer des alias sur des noms de type pour éviter de devoir taper ce long nom de type à chaque fois. Par exemple si l'on souhaite renommer le type précédent en *Iterateur*, on écrit :

Code : C++

```
typedef std::map<std::string, std::vector<int> >::iterator Iterateur
```

A partir de là, on peut déclarer des objets de ce type en utilisant l'alias :

Code : C++

```
Iterateur it;
```

Évidemment, si on utilise qu'une seule fois un objet de ce type, on ne gagne rien, mais dans de longs codes, cela peut devenir pratique.

Vous n'êtes pas convaincu ? Vous trouvez ça inutile ? J'ai un argument en béton pour vous convaincre ! On vous avait révélé la vérité sur les `string` dans le [chapitre d'introduction à la POO](#). Mais on ne vous avait pas tout dit !

`string` est en réalité un alias d'un type template plutôt compliqué. Le fichier d'en-tête `string` de la SL ne contient en réalité pas beaucoup plus que ça :

Code : C++

```
typedef basic_string<char, char_traits<char>, allocator<char> >
string;
```



C'est quand même bien plus simple d'écrire `string` que tout ce long type à chaque fois, non ?

... sur la bibliothèque Qt

Dans le cours, nous avons eu largement le temps d'étudier la bibliothèque Qt et de découvrir à quel point il était simple de créer des GUI (fenêtres).

Nous avons aussi découvert que cette bibliothèque était énorme, et qu'on devait plutôt parler de *framework* (ensemble de bibliothèques).

Je vous rappelle que Qt est constitué de plusieurs modules :

- GUI
- OpenGL
- Dessin
- Réseau
- SVG
- Scripts
- XML
- SQL
- Core

En ce qui nous concerne, nous avons eu l'occasion de bien faire le tour du module GUI (c'était le but !) et nous nous sommes initiés aussi un peu au réseau.

Malgré cela, nous n'avons pas tout vu sur le module GUI. D'autre part, nous avons seulement effleuré le module réseau, et nous n'avons pas du tout parlé des autres modules.

Je vais, dans cette annexe, vous présenter brièvement quelques-uns de ces modules. Je ne vais pas vous les expliquer (ce serait beaucoup trop long !), juste vous en parler pour vous donner quelques pistes.

Surtout, pensez à vous rendre sur [la doc](#) pour en savoir plus ! 😊

Module GUI : des petites fonctionnalités cachées

Il y a quelques widgets et fonctionnalités plus rares dont je n'ai pas eu l'occasion de parler. Je vais vous en présenter quelques-uns rapidement ici. Ils ne sont pas toujours utiles mais ça peut être bien de savoir qu'ils existent.

Cette liste des autres fonctionnalités à découvrir n'est pas complète, loin de là. Je ne connais pas tout. Je vous donne juste une idée des "petites choses" que vous pouvez découvrir si vous passez un peu de temps dans la doc. 😊

QCalendarWidget : un calendrier tout prêt

Le widget `QCalendarWidget` permet d'afficher un calendrier :

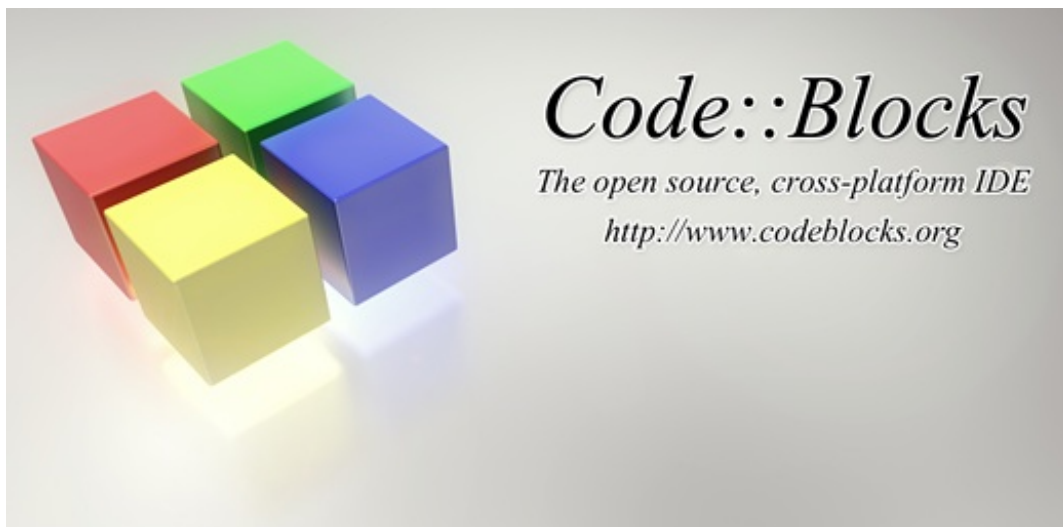


	dim.	lun.	mar.	mer.	jeu.	ven.	sam.
27	29	30	1	2	3	4	5
28	6	7	8	9	10	11	12
29	13	14	15	16	17	18	19
30	20	21	22	23	24	25	26
31	27	28	29	30	31	1	2
32	3	4	5	6	7	8	9

Si vous devez réaliser un agenda ou si l'utilisateur doit sélectionner une date, nul doute que ce widget vous fera gagner un temps fou !

QSplashScreen : pour faire patienter au démarrage

Parfois, certains programmes sont un peu longs à charger. Pour faire patienter l'utilisateur, on affiche un "splash screen", c'est-à-dire une petite image au centre de l'écran. C'est ce que fait Code::Blocks au démarrage par exemple.



Qt permet de créer un "splash screen" avec la classe `QSplashScreen`. On l'utilise en général dans le main, juste avant d'ouvrir la fenêtre principale :

Code : C++

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include <QSplashScreen>
#include <QPixmap>
#include "FenPrincipale.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QSplashScreen splash(QPixmap("znavigo.png"),
Qt::WindowStaysOnTopHint);
    splash.show();

    // Traduction des chaînes prédéfinies par Qt dans notre langue
    QString locale = QLocale::system().name();
    QTranslator translator;
    translator.load(QString("qt_") + locale,
    QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    // Ouverture de la fenêtre principale du navigateur
    FenPrincipale principale;
    principale.show();

    return app.exec();
}
```

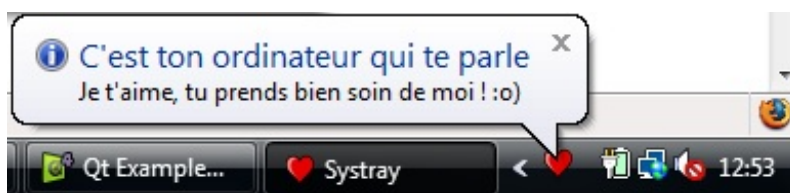
Le splash screen peut être arrêté en cliquant dessus.

Après, libre à vous de l'arrêter automatiquement au bout d'un certain temps, il faut juste chercher dans la doc comment faire.

Afficher une icône dans le system tray

Pour certaines applications résidentes en mémoire, il peut être utile de placer une icône dans le system tray, oui là à côté de l'horloge vous savez. 😊

Qt permet justement de le faire avec QSystemTrayIcon :



Le mieux pour apprendre à s'en servir est de jeter un oeil à l'[exemple fourni dans la doc de Qt](#).

Module réseau : utilisez des classes de haut niveau

Dans notre découverte du réseau, nous avons utilisé des QTcpSocket et un QTcpServer. C'est une gestion assez bas niveau des paquets et il nous a fallu apprendre un peu comment le réseau fonctionnait. On aurait pu parler des paquets UDP aussi, mais on les utilise vraiment dans des cas spécifiques.

En revanche, ce qu'on n'a pas vu, c'est qu'il y a des classes de plus haut niveau qui vous évitent d'avoir à manipuler les paquets TCP directement. Je pense en particulier à :

- **QHttp** : vous permet d'utiliser le protocole HTTP et donc de télécharger des pages web ou des fichiers via le web.
- **QFtp** : vous permet de télécharger et d'envoyer des fichiers par FTP. Vous pourriez créer votre propre client FTP comme Filezilla par exemple. 😊

Ces classes sont beaucoup plus faciles à utiliser que celles que nous avons vues, donc n'hésitez pas à y jeter un oeil. Elles sont brièvement introduites dans la [page d'accueil du module](#) réseau sur la doc de Qt.

Module SQL : accès aux bases de données

Si votre programme doit enregistrer de nombreuses données, il peut être utile de les stocker dans une base de données. C'est un système puissant pour enregistrer des informations, mais il faut connaître le langage SQL pour écrire et lire des informations dedans.

Qt propose tout ce qu'il faut pour se connecter à une base de données dans votre programme, mais il n'inclue pas la base de données... ce sera à vous de l'installer. En clair, si vous utilisez MySQL comme base de données, il faudra d'abord aller installer MySQL sur le [site officiel](#) avant de pouvoir établir une connexion avec dans votre programme.

MySQL est un système de gestion de base de données puissant mais évitez d'y avoir recours systématiquement dans vos programmes. Ce serait un peu utiliser un tank équipé de missiles nucléaires pour tuer une mouche.

Parfois, stocker les meilleurs scores dans un jeu pourrait être facilement fait dans des fichiers (avec QFile par exemple) sous forme de texte simple ou au format XML (je vais en parler un peu plus loin). Inutile de sortir l'artillerie lourde MySQL pour ça.

Si toutefois vous avez vraiment besoin d'une base de données mais que vous ne voulez pas utiliser MySQL qui est un peu gros, jetez un oeil du côté de [SQLite](#) qui est tout léger (mais un peu moins complet).

Une fois que vous avez installé votre système de gestion de base de données sur votre ordinateur, vous pouvez découvrir comment y faire appel depuis Qt. Le mieux est de lire l'[introduction au module QtSql](#) sur la doc. En tout cas c'est ce que je ferais à votre place.

En quelques minutes de lecture de cette seule page, vous devriez déjà savoir vous connecter à la base de données et exécuter des requêtes SQL (mais attention, il faut connaître le langage SQL avant !).

Module XML : pour ceux qui doivent gérer des données au format XML

Le XML est un langage générique qui est à la base de nombreux autres langages, comme XHTML (qui permet de créer des pages web).

Le principe de XML peut être très vite compris si vous avez déjà fait du XHTML avant. En gros, c'est vous qui définissez vos propres balises :

Code : XML

```
<bibliotheque>
  <livre>
    <auteur>J.R.R. Tolkien</auteur>
    <titre>Le seigneur des anneaux</titre>
  </livre>
  <livre>
    <auteur>R. Barjavel</auteur>
    <titre>La nuit des temps</titre>
  </livre>
</bibliotheque>
```

Les données sont placées entre des balises que vous définissez. L'avantage du XML est qu'il est facile à lire (enfin, tant que le fichier n'est pas trop gros ou trop complexe).

Vous pouvez vous servir de cette technique pour organiser vos données dans des fichiers sans avoir recours à une base de données. D'autre part, le XML est un format d'échange devenu courant de nos jours, et il est possible que quelqu'un vous "envoie" des données au format XML que vous devrez traiter dans votre programme.

Pour lire le contenu d'un document XML comme celui ci-dessus (et pour écrire du XML aussi), il y a le module QtXml qui permet de faire cela facilement. Il vous faudra acquérir avant un peu de théorie sur le fonctionnement de XML (DOM, SAX, XQuery, DTD, XML Schema...). Il vaut mieux être rôdé sur la théorie de XML avant de s'y lancer sinon vous n'en profiterez pas. 😊



Je vous conseille de lire cette [petite introduction à XML](#) sur le Site du Zéro avant de faire des recherches plus approfondies.
Wikipédia est une bonne source de départ aussi.

Une fois que vous connaissez un peu mieux le fonctionnement de XML, direction la [page d'accueil du module QtXml](#) pour découvrir les outils que Qt met à votre disposition pour lire et écrire du XML. Il y a de quoi faire, et encore une fois je vous le rappelle, mieux vaut être armé et connaître XML avant de se lancer là-dedans !

Module Core : toutes les fonctionnalités de base de Qt

Le module QtCore contient des classes de base de Qt qui n'ont pas de rapport avec les GUI et qui peuvent donc être utilisées dans un programme purement console.

Dans ce module, on trouve un certain nombre de classes que vous connaissez déjà :

- **QString** : gestion des chaînes de caractères.
- **QByteArray** : une suite d'octets (on s'en est servi dans le programme de Chat pour construire des paquets).
- **QFile** : accès aux fichiers.
- **QLocale** : permet d'accéder aux habitudes de représentation des nombres et chaînes dans différentes langues.
- **QList** : une liste capable de stocker un tableau à taille dynamique (cette classe est une version "Qt" de ce qui se fait dans

la STL dont je vous ai parlé plus haut).

- **QUrl** : représente une URL.

Voilà quelques exemples de classes du module QtCore que vous avez déjà utilisées. Comme vous le voyez, ces classes font partie du "cœur" de Qt et pas du module GUI car elles peuvent être réutilisées dans tous les autres modules.

Jetez donc un oeil à la [liste des classes du module QtCore](#). Il y a de quoi faire, et on retrouve notamment de nombreuses versions "Qt" de classes présentes dans la STL (il y a même un [QVector](#) !).

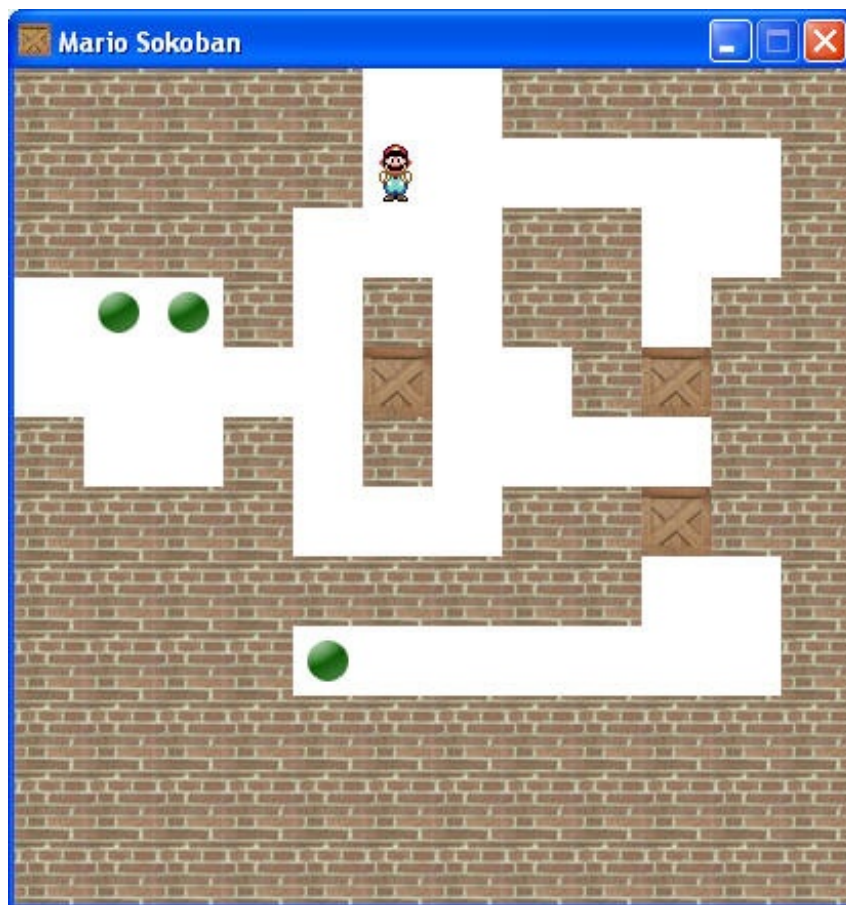
Bonne pêche !

D'autres bibliothèques

Vous en avez fait l'expérience dans ce cours avec Qt, on utilise souvent des bibliothèques externes en C++. Le problème c'est qu'il y en a des milliers et que l'on ne sait pas forcément laquelle choisir. Tenez, rien que pour créer des fenêtres, je pourrais vous citer une dizaine de bibliothèques performantes. Heureusement, je suis là pour vous aider un peu dans cette jungle.

Créer des jeux en 2D

Si vous avez lu le [cours de C](#), vous avez certainement appris à utiliser la bibliothèque SDL pour créer des jeux en 2D, comme par exemple le "Mario Sokoban" :



On peut tout à fait utiliser la SDL en C++, mais il existe d'autres bibliothèques utilisant la force de la programmation orientée objet qui sont plus adaptées à notre langage favori.

Allegro

[Allegro](#) est une bibliothèque multi-plateforme dédiée aux jeux vidéos. Ses créateurs ont particulièrement optimisé leurs fonctions de sorte à ce que les jeux réalisés soient aussi rapides que possible. Elle gère tout ce qui est nécessaire à la création d'un jeu, les joysticks, le son, les images, les boutons et autres cases

Allegro

à cocher. Son principal défaut, pour nous francophones, est que sa documentation est anglais.



La SFML

La SFML se décrit elle-même comme étant une alternative orientée objet à la SDL.

Cette bibliothèque est très simple d'utilisation et propose également tous les outils nécessaires à la création de jeux sous forme de classes. Un autre avantage est qu'elle est découpée en petits modules indépendants, ce qui permet de n'utiliser que la partie dédiée au son ou que la partie dédiée à la communication sur le réseau par exemple.

Finalement, tout est [documenté](#) en français et son créateur, [Laurent Gomila](#), passe souvent sur les [forums du SdZ](#) pour aider les débutants. C'est donc un bon choix pour débiter dans le domaine passionnant des jeux vidéos.



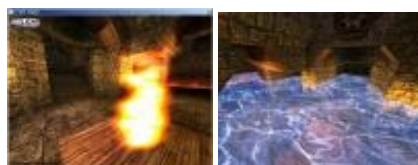
Faire de la 3D

Encore un domaine très vaste et très intéressant. De nos jours la plupart des jeux vidéos sont réalisés en 3D et beaucoup de monde se lance dans la programmation C++ justement dans le but de réaliser des jeux en trois dimensions. De base, il existe deux APIs pour manipuler les cartes graphiques : DirectX et OpenGL, la première n'étant disponible de base que sous Windows. Vous avez certainement déjà dû entendre ces deux noms. 😊

Avec ça, on peut tout faire, tout dessiner, tout réaliser. Le problème, c'est que ces deux APIs ne proposent que des fonctionnalités de base comme dessiner un triangle ou un point. Réaliser une scène complète avec un personnage qui bouge et des animations demande donc beaucoup de travail. C'est pour cela qu'il existe ce qu'on appelle des "moteurs 3D" qui proposent des fonctionnalités plus haut-niveau et donc plus simples à utiliser. Tous les jeux vidéos que vous connaissez utilisent des moteurs 3D, c'est la vraie boîte à outils qu'utilisent les programmeurs.

Parmi tous les moteurs existants, je vous en cite deux bien connus et simples d'utilisation : [Irrlicht](#) et [Ogre3D](#).

Ces deux bibliothèques proposent globalement le même lot de classes et fonctions. Comme bien souvent, les documentations de ces moteurs sont en anglais, mais vous avez de la chance, il existe sur le SdZ deux cours d'introduction à ses outils. Vous les trouverez [ici](#).



Pour choisir entre ces deux bibliothèques (ou parmi d'autres encore), je vous conseille de regarder quelques codes sources d'exemple et le début des cours d'introduction. Vous serez alors plus à même de décider lequel vous plaît le plus.

Plus de GUI

Vous avez appris à utiliser Qt dans ce cours, mais il n'y a bien sûr pas que ce framework pour réaliser des applications avec des fenêtres. En fait, le choix est gigantesque ! Je vais ici vous présenter brièvement deux bibliothèques que l'on voit dans de nombreux projets.

wxWidgets

wxWidgets ressemble beaucoup à Qt dans sa manière de créer les fenêtres et les widgets qui s'y trouvent. On y retrouve aussi la notion de signaux, de slots et de connexions entre eux. Vous devriez donc facilement vous y retrouver. L'éditeur Code::Blocks que nous utilisons depuis le début du cours est, par exemple, basé sur wxWidgets. On peut donc réaliser de belles choses. 😊



.NET (prononcez "dot net") est le framework de création de fenêtre développé par Microsoft. En plus des fonctionnalités liées aux GUIs, .NET permet d'interagir complètement avec Windows et permet d'accéder à de nombreux services comme la communication sur le réseau ou la gestion du son. Bref, c'est une bibliothèque vraiment

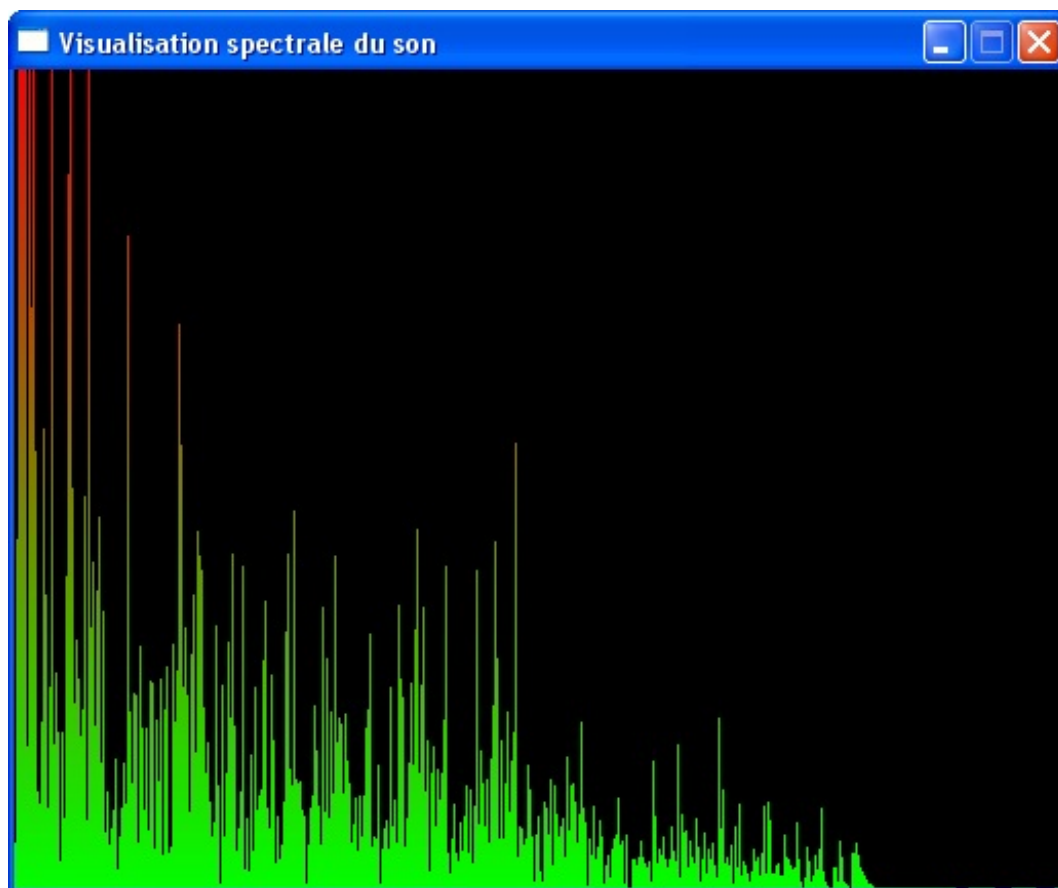


très complète. Presque tous les logiciels que vous connaissez sous Windows l'utilisent aujourd'hui. C'est vraiment un outil incontournable. De plus, elle est très bien documentée, ce qui permet de trouver rapidement et facilement les informations nécessaires. Son seul défaut est qu'elle n'est disponible entièrement que sous Windows. Il existe des projets comme [Mono](#) qui tentent d'en proposer une version sous Mac et Linux, mais tout n'est pas encore disponible.



Manipuler du son

Alors là, c'est plus simple de faire son choix. Il y a bien sûr beaucoup de bibliothèques qui permettent de manipuler du son, mais il y en a une qui écrase tellement la concurrence que je vais m'y limiter. Il s'agit de [FMOD EX](#). Presque tous les jeux vidéos que vous connaissez l'utilisent, c'est dire !



Cette bibliothèque permet de lire à peu près tous les formats de fichiers sonores, du wav au mp3, tout y est. On peut ensuite jouer ces sons, les transformer, les filtrer, les distordre, y ajouter des effets, etc. Il n'y a presque aucune limite. Je crois que vous l'avez compris, c'est le choix à faire dans le domaine.

Boost

Je ne pouvais pas terminer ce chapitre sans vous parler de [boost](#). C'est la bibliothèque incontournable de ces dernières années. Elle propose près d'une centaine de modules dédiés à des tâches bien spécifiques. On peut vraiment la voir comme une extension de la STL. Chaque module de boost a été écrit avec grand soin souvent dans le cadre de recherche en informatique. C'est donc un vrai gage de fiabilité et d'optimisation.



Je ne peux pas vous présenter ici tout ce qu'on y trouve. Il me faudrait pour ça, un deuxième tutoriel au moins aussi long que celui-là. 😊 Mais je vous invite à jeter un oeil à la [liste complète des fonctionnalités](#).

En résumé, on y trouve :

- De nombreux outils mathématiques (générateurs aléatoires, fonctions compliquées, matrices, nombres hyper-complexes, outils pour les statistiques, ...).
- Des pointeurs intelligents. Ce sont des outils qui gèrent intelligemment la mémoire et évitent les problèmes qui

- surviennent quand on manipule dangereusement des pointeurs.
- Des outils dédiés à la communication sur le réseau.
- Des fonctions pour la mesure du temps et de la date.
- Des outils pour naviguer dans l'arborescence des fichiers.
- Des outils pour la manipulation d'images de tout format.
- Des outils pour utiliser plusieurs coeurs d'un processeur dans un programme.
- Des outils pour exécuter un code source [python](#) en C++.
- ...

Vous voyez, il y a vraiment de tout. La plupart des fonctionnalités sont proposées sous forme de templates et donc entièrement optimisées pour votre utilisation lors de la compilation. C'est vraiment du grand art ! Je ne peux que vous recommander d'user et même d'abuser de boost. 😊

Comme je vous l'ai dit, cette liste n'est bien sûr pas complète. L'important est de choisir un outil avec lequel vous vous sentez à l'aise. N'hésitez pas à surfer sur le web pour trouver d'autres options ou d'autres utilisateurs qui présentent leurs préférences. Vous pouvez aussi poser des questions sur le [forum C++](#) du SdZ. La communauté se fera un plaisir de vous répondre et de vous guider dans vos choix.

J'espère que cette annexe aura rempli son rôle : vous aider à regarder dans de nouvelles directions. L'inconnu, ça fait un peu peur au début, mais on s'y fait très vite vous verrez. 😊

Comme vous avez pu le voir, tout ce que vous pouvez faire en C++ (et en programmation en général) est tellement riche qu'on n'aurait jamais assez d'une vie pour tout connaître. J'espère que vous me comprenez maintenant. 😊

Plutôt que de tout apprendre, essayez plutôt de découvrir une nouvelle notion à la fois. Si vous vous éparpillez trop, vous aurez du mal à bien assimiler ces connaissances.

Bon courage, et bonne continuation ! 😊

Le cours de C++ s'arrête là !

J'espère que vous aurez appris au moins autant de choses que vous ne l'espériez, et surtout que vous avez formé votre esprit à être capable de programmer en toutes circonstances par la suite.

N'hésitez pas à lire le dernier chapitre "Ce que vous pouvez encore apprendre", qui vous donne de nombreuses ouvertures pour continuer votre apprentissage si vous le désirez. 😊